COMMODORE 64
CODICE MACCHINA

Come ottenere maggiore velocità e potenza

162 255 160 100 136

208 253 202 208 248

IAN SINCLAIR

tecniche nuove

Ian Sinclair

COMMODORE 64
CODICE MACCHINA

Ian Sinclair

COMMODORE 64
CODICE MACCHINA

Indice

Prefazione

Introduzione al linguaggio macchina del Commodore 64

Se molte persone che utilizzano il computer si accontentano di programmare in BASIC, molte altre desiderano apprendere più di quanto la semplice programmazione in BASIC permetta di fare. Pochi, tuttavia, sono coloro che approfondiscono il linguaggio macchina che permette di avere un maggiore controllo sul computer stesso. La ragione di tutto questo è, a mio avviso, da attribuirsi al fatto che troppi libri che trattano l'argomento partono dal presupposto che il lettore abbia già familiarità con il linguaggio tecnico relativo a questo tipo di programmazione. Inoltre, la maggior parte dei libri sulla programmazione in linguaggio macchina tratta la materia come uno studio a sé stante, dando poche indicazioni riguardo le sue applicazioni.

Questo libro ha un duplice scopo: 1) introdurre chi possiede il Commodore 64 al funzionamento del suo computer, permettendo di ottenere una programmazione più efficace senza dovere per forza ricorrere al linguaggio macchina; 2) introdurre la programmazione in linguaggio macchina in modo semplice. Devo soffermarmi sulla parola 'introdurre' poiché nessun libro che tratta il linguaggio macchina può dirvi tutto, anche nel caso in cui illustrasse il linguaggio macchina di un singolo computer.

Ciò che posso affermare è di potervi iniziare in proposito, il che significa che sarete in grado di scrivere brevi programmi in linguaggio macchina, comprendere quelli che appaiono sulle riviste specializzate e, in generale, aumentare le possibilità d'uso del vostro Commodore 64. Potrete inoltre far uso di testi che approfondiscono ulteriormente la materia aumentando le vostre capacità in questo affascinante campo.

La conoscenza del sistema operativo del Commodore 64 e la possibilità di programmare in linguaggio macchina vi aprirà nuovi orizzonti nella programmazione. Ecco perché la maggior parte dei giochi vengono scritti in linguaggio macchina. Troverete anche che molti programmi scritti principalmente in BASIC includono subroutine in linguaggio mac-

china permettendo una più elevata rapidità di esecuzione e un maggiore controllo sulla macchina stessa.

I miei ringraziamenti più sentiti vanno a coloro che hanno reso possibile la realizzazione del presente libro. Fra questi, Richard Miles della casa editrice Granada che me lo ha commissionato fornendomi un computer Commodore 64 con tanto di stampante. Egli e Sue Moore, anch'ella della Granada, hanno fatto miracoli lavorando meticolosamente sul mio manoscritto. I miei ringraziamenti vanno anche a Milton Bathurst, che non ho mai incontrato personalmente pur avendo consultato ed apprezzato molto il suo libro *Inside The Commodore 64*.

Ian Sinclair

Capitolo 1

ROM, RAM, byte e bit

Una delle cose che scoraggiano l'utente di un computer dal tentare di andare oltre il BASIC è il numero di parole nuove che si incontrano. A riguardo, molti scrittori sembrano dare per scontato, soprattutto per quanto riguarda il linguaggio macchina, che il lettore abbia una conoscenza approfondita di elettronica tale da poter comprendere tutti i termini usati. Per quanto mi riguarda, partirò dal presupposto che voi non abbiate questa conoscenza, ma che possediate un Commodor 64 e una certa esperienza nella programmazione in linguaggio BASIC. Ciò significa che cominceremo proprio dall'inizio. Non avendo l'intenzione di interrompere importanti spiegazioni con dettagli tecnici o matematici, ho rimandato la loro visione alle appendici in fondo al libro. In tal modo, se lo vorrete potrete dare uno sguardo ai dettagli che vi interessano, altrimenti potrete saltarli tranquillamente.

Per iniziare dobbiamo pensare alla memoria. Per quanto ci riguarda, l'unità fondamentale nella memoria del computer è un circuito elettrico a mo' di interruttore. Quando si entra in una stanza e si accende la luce, nessuno pensa che essa rimanga accesa fino a quando l'interruttore non viene disattivato, eppure è così. Con questo, non andate a dire agli amici che un circuito elettrico contiene una memoria. Ora la memoria di un computer possiede tantissimi interruttori miniaturizzati che possono essere accesi o spenti. Ciò che li rende una memoria è il fatto che essi rimangono nelle condizioni in cui sono stati attivati fino a quando il loro stato non viene alterato.

Una unità di memoria, come questa nel computer, viene chiamata bit, abbreviazione di binary digit (numero binario), e ha due possibili stati (ON-OFF).

Rimarremo ancorati all'esempio dell'interruttore poiché si rivela utilissimo. Sopponete di dovere comunicare tramite segnali elettrici usando degli interruttori. Si potrebbe far uso di un circuito come nella figura 1.1. Quando l'interruttore è attivato e la luce è accesa, chiamiamo il relativo segnale "SI". Quando l'interruttore è disattivato e la luce è spenta, chiamiamo il relativo segnale "NO". Potreste attribuire, a piacere,

qualsiasi coppia di significati a questi due tipi di condizioni (chiamati "stati") della luce, purché ce ne siano solo due.

Le cose cambiano se si usano due interruttori e due lampadine come nella figura 1.2. In questo caso sono possibili 4 combinazioni diverse: a) entrambe spente, b) A accesa, B spenta; c) A spenta, B accesa; d) entrambe accese. Ciò significa che possiamo trasmettere 4 segnali differenti a seconda del tipo di combinazione di luci che usiamo. Usando una luce abbiamo due combinazioni, usando due luci ne abbiamo quattro. Se ne usassimo tre scopriremmo che le combinazioni salirebbero a otto. Così, dato che 4 è dato dal prodotto 2x2 mentre 8 da $2 \times 2 \times 2$, allora 4 luci permetteranno $2 \times 2 \times 2 \times 2$ combinazioni, vale a dire 16. Essendo vera la cosa e siccome, d'abitudine, il prodotto $2 \times 2 \times 2 \times 2$ viene scritto 2^4 (2 elevato alla quarta potenza), si può facilmente calcolare quante combinazioni si hanno conoscendo il numero di luci a disposizione. Otto luci permetteranno 2^8 combinazioni, cioè 256. Un set formato da otto interruttori potrebbe essere predisposto per trasmettere 256 segnali differenti. Spetterà a noi decidere come usare questi segnali.

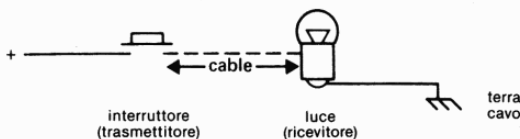



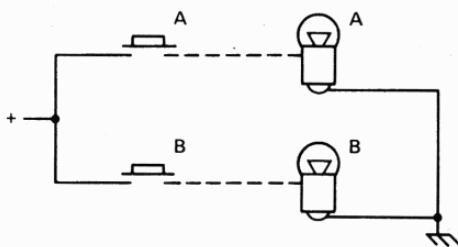
Figura 1.1 - Sistema che usa una sola lampadina permettendo due tipi di segnale differenti

A	B
off	off
off	on
on	off
on	on

Figura 1.2 - Usando 2 lampadine si possono trasmettere quattro tipi di segnali diversi

Un metodo particolarmente utile è il codice binario, tramite il quale si possono scrivere tutti i numeri usando i soli due simboli 0 e 1. Al simbolo 0 si può associare l'idea di "spento", mentre all'1 quella di "acceso". Usando 8 interruttori si possono formare 256 numeri diversi composti da 0 ("spento") e 1 ("acceso"). Questo gruppo di otto interruttori viene chiamato byte, ed è la grandezza usata per calcolare le dimensioni della memoria del computer. Ciò spiega anche come mai i numeri 8 e 256 ricorrano tanto nella programmazione in linguaggio macchina.

Il modo in cui i bit sono sistemati all'interno di un byte per indicare un numero è identico a quello usato per indicare un normalissimo numero. Quando si scrive un numero come 256, il 6 significa sei unità, il 5 che viene subito a sinistra cinque decine, mentre il 2 significa due centinaia. La posizione assunta dalla cifre determina la loro importanza (vedi figura 1.3). Nel 256 il 6 viene chiamato cifra meno importante o meno «significativa», mentre il 2 viene chiamato cifra più importante o più «significativa». Se nel numero 256 si cambia il 6 in 7 o 5, si cambia di una sola unità il numero, mentre se è il 2 ad essere cambiato in 1 o in 3, allora il numero cambia di un centinaio, quantità molto più "significativa".

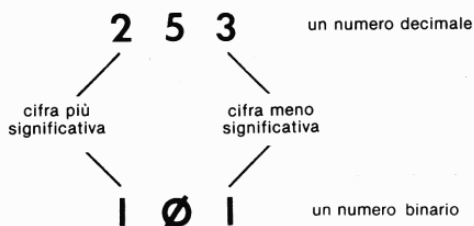



Figura 1.3 - Cifre significative. Il nostro sistema di numerazione fa uso della posizione di una cifra per indicarne il significato

Dopo avere analizzato i bit e i byte, è utile tornare all'esempio della memoria come insieme di interruttori. All'interno del computer sono necessari due tipi di memoria. Il primo deve essere permanente, come gli interruttori meccanici o le connessioni fisse, poiché deve essere utilizzato per conservare le istruzioni codificate tramite numeri che fanno funzionare il computer. Questo è il tipo di memoria chiamata ROM, cioè memoria a sola lettura (Read-Only Memory). Ciò significa che si possono estrarre notizie da essa, ma che non si possono cambiare o cancellare. La ROM è la parte più importante del computer, poiché contiene tutte le istruzioni necessarie per svolgere le funzioni BASIC.

Quando scrivete un vostro programma, il computer lo memorizza sotto forma di codici numerici in una parte della memoria che potrà essere utilizzata ripetutamente. Questa è un tipo di memoria diversa dalla prima, che può essere "scritta" e "letta" e che, a rigor di logica, dovrebbe essere chiamata RWM, cioè memoria a scrittura-lettura. Purtroppo, non seguendo questa logica, si preferisce chiamarla RAM, cioè memoria ad accesso casuale (Random Access Memory). Tale era il nome usato agli inizi dell'era dei computer per distinguere questo tipo di memoria da un'altra che operava in modo diverso. Ci siamo abituati ad usare il nome RAM e probabilmente lo manterremo per sempre!

I codici numerici

Torniamo ora ai byte. In precedenza abbiamo visto che gli otto bit che formano un byte possono assumere 256 diverse configurazioni. Il metodo migliore è quello di rappresentarle tramite il sistema di numerazione binario, partendo dallo 0 per arrivare al 255 (e non dall'1 al 256 poiché è necessario avere una rappresentazione dello 0). Ciascuno dei 38911 byte di RAM disponibili sul Commodore 64 può memorizzare un numero da 0 a 255.

Di per se stessi i numeri non sono di grande utilità, e non potremmo utilizzare adeguatamente un computer se esso potesse trattare solo i numeri da 0 a 255: li si utilizza come codici.

In effetti, ogni codice potrebbe essere usato per svariati scopi. Se avete lavorato utilizzando il codice ASCII in ambiente BASIC saprete che ogni lettera dell'alfabeto, i numero da 0 a 9 e i segni di punteggiatura sono codificati in ASCII utilizzando i numeri che vanno da 32 (lo spazio) a 95 (freccia sinistra). Questa scelta lascia un largo numero di codici ASCII utilizzabili per altri scopi come, ad esempio, la grafica. Tuttavia il codice ASCII non è l'unico. Il Commodore 64 utilizza questi numeri per codificare, ad esempio, l'istruzione PRINT. Quando, infatti, voi digitate la parola PRINT sul Commodore 64 essa non viene memorizzata tramite il codice ASCII (che sarebbe 80,82,73,78,84, cioè un byte per ogni lettera), ma utilizza un solo byte col numero 153 binario. Questo singolo byte viene chiamato TOKEN e può essere utilizzato in due modi. Il primo potrebbe essere quello di individuare i codici ASCII dei caratteri che formano la parola PRINT. Questi sono contenuti nella ROM, così che listando un programma, vedrete apparire la parola PRINT e non il carattere il cui codice è 153. L'altro uso più importante del TOKEN è quello di individuare un set di istruzioni conservato anch'esso nella ROM sotto forma di codici numerici. Queste istruzioni faranno

sì che i caratteri appaiano sullo schermo. I numeri che formano questi codici sono quelli che formano il linguaggio macchina. Essi hanno il controllo diretto delle operazioni all'interno del computer.

Il controllo diretto è la ragione per cui vogliamo usare il linguaggio macchina.

In ambiente BASIC i soli comandi utilizzabili sono quelli per cui sono stati predisposti i TOKEN. Usando il linguaggio macchina, possiamo inventare comandi personalizzati per gli scopi che più ci interessano.

Per inciso, il fatto che PRINT generi un TOKEN è la ragione per cui è possibile usare ? al posto di PRINT. Il Commodore 64 è stato progettato in modo tale da immettere in memoria il codice 153 anche quando un punto interrogativo non venga posto tra virgolette.

Un piccolo programma

Per comprendere meglio quanto è stato detto, provate il seguente programma (fig. 1.4). Esso è stato pensato per mostrarvi le parole chiave presenti nella ROM tramite l'istruzione PEEK del BASIC.

```
10 PRINT41118;" ";:FOR N=41118 TO 41373
20 K=PEEK(N)
30 IF K<128 THEN PRINT CHR$(K);
40 IFK>=128THENPRINTCHR$(K-128):PRINTN+1;" ";
50 NEXT
```

Figura 1.4 - Programma che mostra le parole chiave del BASIC del Commodore 64

PEEK deve essere seguito da un numero o da una variabile numerica chiusa fra virgolette, e significa: "Cerca il valore del byte immagazzinato in questo indirizzo". Ogni byte di memoria del vostro Commodore 64 è numerato, partendo dallo 0, tramite un numero intero positivo. Poiché questa numerazione assomiglia del tutto alla numerazione delle case di una via, ci si riferisce a questi numeri chiamandoli INDIRIZZI. PEEK ricerca il numero, compreso tra 0 e 255, contenuto in ogni indirizzo. Il Commodore 64 converte automaticamente questi numeri dal modo in cui sono memorizzati (forma binaria), nella forma decimale comune, da noi normalmente usata. Tramite l'uso di CHR$ nel nostro programma, si possono stampare i caratteri il cui codice ASCII è il numero ispezionato dalla funzione PEEK. Il programma utilizza la variabile N co-

me numero dell'indirizzo, quindi si accerta che PEEK (N) restituisca un numero inferiore a 128, in altre parole un numero che appartenga al codice ASCII.

Se così accade, allora esso viene stampato.

Ora, la ragione del nostro controllo è che l'ultimo carattere in ogni gruppo di parole, o in ogni parola, è codificato in modo diverso. Al numero che ricaviamo per l'ultimo carattere, è stato aggiunto 128 nel codice ASCII. Ad esempio, le prime tre locazioni ispezionate da PEEK contengono i numeri 69, 78 e 196. Il numero 69 è il codice ASCII che sta per E, 78 è il codice di N, quindi 68 (196—128) è il codice di D). Questa è la locazione in cui è contenuta la parola END. La ragione per cui quest'ultima lettera viene trattata così diversamente è il risparmio di memoria! Se si lasciasse uno spazio tra le parole, ciò significherebbe un byte di memoria sprecato. Nel modo suddetto non c'è spreco perché l'ultima lettera di un gruppo ha sempre un numero di codice superiore a 128, permettendo al computer di riconoscerlo facilmente.

Abbiamo seguito il medesimo schema, per il programma BASIC che appare nella figura 1.4, usando nella linea 40 l'istruzione PRINT per stampare la lettera corretta, per scegliere la linea nuova e stampare il numero dell'indirizzo. Esiste un altro set di numeri memorizzati precedentemente, il quale è formato da altri indirizzi. Questi sono gli indirizzi delle subroutine che eseguono i comandi BASIC e sono memorizzati nello stesso ordine delle parole.

Lo spaccato del Commodore 64

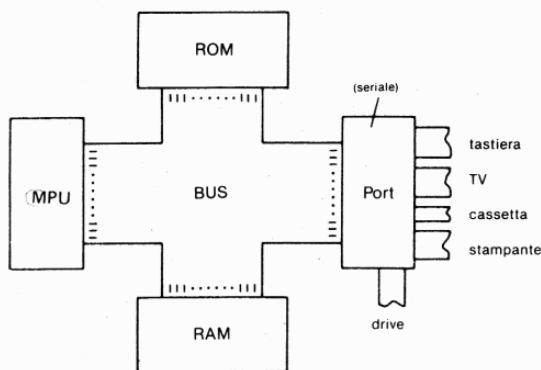



Figura 1.5 - Il diagramma a blocchi del Commodore 64. Le connessioni, chiamate bus, sono formate da vari legami che uniscono le unità del sistema

Date un'occhiata allo schema della figura 1.5. Sebbene sia semplice poiché ho tralasciato i particolari, è sufficiente per fornirci una prima idea della struttura interna del Commodore 64. Questo tipo di schema viene chiamato diagramma a blocchi, poiché ogni unità è rappresentata da un blocco senza che ne compaia il contenuto. I diagrammi a blocchi sono come delle cartine in scala che ci mostrano le strade principali che collegano le città, ma che non mostrano quelle secondarie o le vie cittadine. Un diagramma a blocchi è sufficiente per indicarci i principali sentieri percorsi dai segnali elettrici all'interno del computer. Due di questi blocchi sono la memoria ROM e quella RAM, che già conoscete.

Un altro blocco è rappresentato dalla CPU (unità di elaborazione centrale — Central Processing Unit —), chiamata anche MPU (Microprocessor Unit). La CPU è l'unità più importanta del sistema ed è, a tutti gli effetti, una unità a sè stante. La CPU è un unico blocco estraibile, un chip al silicio, posto in un involucro di plastica nera e provvisto di 40 piedini disposti su due file da 20 (fig. 1.6).

Esistono diversi tipi di CPU prodotti da fabbricanti diversi. Quello montato sul Commodore 64 è chiamato 6502 (o 6502A).

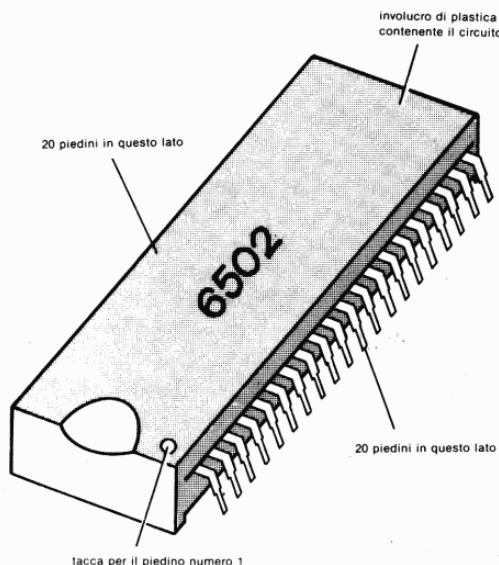



Figura 1.6 - Il microprocessore 6502. La parte che in realtà funziona è più piccola di un'unghia, mentre l'involucro plastico nero (di 52×14 mm) permette di lavorarvi più facilmente

Che funzioni svolge la CPU? Praticamente tutto, e tuttavia gli interventi che essa può svolgere sono straordinariamente pochi e semplici. La CPU può "caricare" un byte, cioè può copiare il contenuto di un byte dalla memoria e immagazzinarlo al suo interno. La CPU può anche compiere l'operazione inversa, cioè memorizzare in un qualsiasi indirizzo della memoria un byte precedentemente copiato dalla memoria stessa.

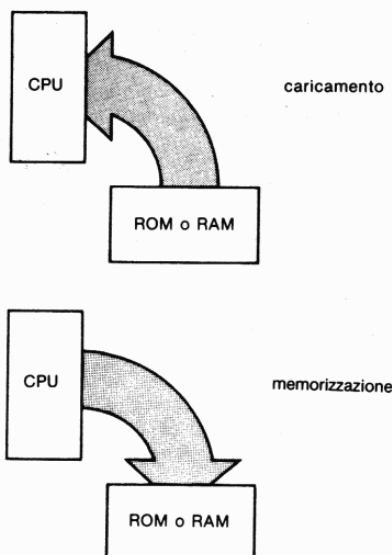



Figura 1.7 - Caricamento e memorizzazione. Per caricamento si intende segnalazione alla CPU dalla memoria, così che le cifre di un byte possano essere copiate all'interno della CPU. Per memorizzazione si intende il procedimento opposto

Il microprocessore impiega la maggior parte del suo tempo ad espletare questi due interventi (vedi fig. 1.7). Combinandoli insieme si possono copiare byte da una parte della memoria all'altra. Non credete che ciò sia utile? Ebbene, questo è quanto succede quando premete il tasto della lettera H sulla tastiera e vedete che essa appare sullo schermo. La CPU tratta la tastiera come una parte di memoria e il video come un'altra, e copia i byte dalla prima verso l'altra. Sebbene questa sia una grossa esemplificazione, per ora andrà bene ugualmente per illustrare l'importanza di questo intervento.

Il caricamento e la memorizzazione sono due interventi molto importanti della CPU, ma ne esistono altri: uno di questi è il set di operazioni

matematiche. Queste consistono, per la maggior parte della CPU, solo nella somma e nella sottrazione, tramite l'uso di numeri di un solo byte. Siccome un numero di un solo byte significa un numero compreso tra lo 0 e il 255, come potrà il calcolatore svolgere calcoli come moltiplicazioni tra grandi numeri, divisioni, elevazioni a potenza, logaritmi, seni, ecc.? La risposta sta nei programmi in linguaggio macchina memorizzati all'interno della ROM. Se questi programmi non fossero presenti in memoria li dovreste scrivere voi, ma pochi sono coloro che desiderano avere a che fare con questo compito ingrato.

È anche presente, nella CPU, un set di operazioni logiche. La logica della CPU è, come tutte le sue operazioni, semplice e soggetta a rigide regole. Le funzioni logiche confrontano i bit di due byte e formulano una risposta che dipende dal valore dei bit confrontati e dalla regola logica seguita. Le tre regole logiche principali sono chiamate AND, OR e XOR, che la figura 1.8 mostra nella loro applicazione.

Un altro set di interventi è chiamato set di salto. Un salto significa un cambiamento di indirizzo, simile all'istruzione GOTO in BASIC. L'unione tra un salto e una decisione è il modo in cui la CPU espleta i suoi salti decisionali, proprio come si può fare in un programma BASIC:

```
IF A=36 THEN GOTO 1050.
```

In tal modo alla CPU si può fare eseguire un'istruzione presente in un indirizzo che non sia quello immediatamente successivo.

La CPU è un dispositivo programmato che esegue ogni intervento dopo avere copiato un byte di istruzione memorizzato nella memoria. Normalmente, quando la CPU copia un'istruzione da un byte presente in memoria (di solito ROM), la esegue per poi passare all'istruzione presente nel byte successivo. Un'istruzione di salto impedirebbe che ciò accada facendo sì che la CPU legga il contenuto dell'indirizzo specificato nell'istruzione di salto. Inoltre si può fare dipendere il salto dal risultato di un test. Il test verrà di norma portato a termine sulla base del risultato precedente a seconda che, ad esempio, il risultato sia negativo, nullo o positivo.

Gli interventi che la CPU può compiere non sono tanti e quelli che ho omesso non sono molto dissimili da quelli descritti precedentemente, oppure per ora poco importanti. Ciò che voglio sottolineare è che il microprocessore non è poi un dispositivo così fantastico. Ciò che lo rende importante per il computer è che può essere programmato e che può eseguire le operazioni velocemente. Di uguale importanza è il fatto che il microprocessore può essere programmato inviandogli segnali elettrici.

Questi segnali vengono inviati mediante 8 piedini, che rappresentano

AND

Il risultato di AND è un 1 se entrambi i bit sono 1, viceversa è 0:

1 AND 1 = 1 { 1 AND 0 = 0 / 0 AND 1 = 0 } 0 AND 0 = 0

Per due byte, i bit corrispondenti vengono calcolati con la funzione AND:

```
        10110111
AND     00001111
        --------
        00000111
             ---
          soltanto questi bit esistono in entrambi i byte
```

OR

Il risultato di OR è 1 se uno dei due valori o entrambi sono 1, viceversa è 0:

1 OR 1 = 1 { 1 OR 0 = 1 / 0 OR 1 = 1 } 0 OR 0 = 0

Per due byte, i bit corrispondenti vengono calcolati con la funzione OR:

```
        10110111
OR      00001111
        --------
        10111111

         ↑
         il solo bit che è 0 in entrambi i byte
```

XOR (Exclusive — OR)

Si comporta come OR, ma dà uno 0 se i bit sono identici:

1 XOR 1 = 0 { 1 XOR 0 = 1 / 0 XOR 1 = 1 } 0 XOR 0 = 0

```
        10110111
XOR     00001111
        --------
        10111000
        --------
    Se due bit sono identici, il risultato è 0.
```

Figura 1.8 - Le regole delle tre operazioni logiche AND, OR, XOR

il bus dei dati della CPU. Non ci vuole molto per capire che questi 8 piedini rappresentano gli otto bit che formano un byte.

Ogni byte di memoria può alterare la CPU mandandole i suoi segnali elettrici. Chiameremo "lettura" il processo appena descritto per esteso. Lettura significa che un byte di memoria è collegato tramite 8 linee alla CPU in modo tale che ogni bit corrisponde a un segnale che arriverà a un piedino appartenente al bus dei dati. Così come la lettura di un disco o di un foglio di carta non cancella ciò che è inciso o scritto, la lettura della memoria non porta alcuna alterazione alla memoria stessa. Il procedimento opposto (la scrittura) cancella quanto era presente precedentemente in memoria. Quando la CPU scrive un byte in una locazione di memoria, ciò che prima era presente in quella determinata locazione viene perso una volta per tutte, essendo stato rimpiazzato dal nuovo byte. Ecco perché è così facile scrivere delle nuove linee in BASIC sostituendo le vecchie con lo stesso numero di linea.

La tavola d'hôte

Anche se conoscete la programmazione in linguaggio BASIC non siete in grado di scrivere programmi che utilizzino le istruzioni che nella realtà la CPU esegue ogni volta che si esegue un programma. Tutto quello che saprete fare è scrivere un programma in linguaggio BASIC in cui compaiono delle istruzioni in un certo ordine, da voi stabilito, nella speranza che esse eseguano ciò che desideravate. La vostra scelta si limita però alle parole chiave contenute all'interno della ROM. Poiché questa parte della memoria non può essere alterata, se desiderate portare a termine delle operazioni non annoverate tra quelle eseguibili dalle parole chiave del BASIC, avete due possibili alternative: combinare più parole chiave insieme in modo tale da ottenere l'effetto desiderato, o istruire direttamente la CPU usando codici numerici (linguaggio macchina). Consideriamo una situazione che possa illustrare questo paradosso.

Supponete di voler costruire un muro: potreste mandare a chiamare un operaio. Dovrete solamente dirgli che volete costruire un muro attraverso il giardino retrostante, quindi sedervi ed aspettare. Ciò corrisponde all'uso in BASIC di una istruzione del tipo "Costruisci un muro". Tutto il lavoro viene svolto da questa istruzione per cui non dovrete assolutamente preoccuparvi di nulla.

Ora provate a pensare ad un'altra possibilità. Supponete di avere un robot che può eseguire solo delle istruzioni molto semplici, ma in maniera incredibilmente veloce. Non potreste dirgli di costruire un muro, perché questo comando sarebbe fuori dalla sua portata. Dovrete dirglie-

lo nei minimi dettagli come, ad esempio: “traccia una linea lunga 30 metri a partire dal limite della cucina della casa, misurata lungo il recinto in direzione sud, verso un punto distante 32 metri a partire dalla fine del soggiorno sempre misurato lungo il recinto verso sud. Scava una fossa profonda 30 cm e larga 35 lungo il tracciato della linea. Impasta 3 sacchi di sabbia con 2 di cemento e 4 carriole di ghiaia per tre minuti. Versa dell’acqua nell’impasto fino a quando un secchio riempito con esso non impieghi 10 secondi per svuotarsi se tenuto capovolto. Riempi la fossa con l’impasto”. Le istruzioni sono molto dettagliate (e lo devono essere per un robot così sciocco), ma tali da essere eseguite con precisione e velocemente. Se avete dimenticato qualcosa, non importa quanto ovvia sia, essa non verrà eseguita. Se dimenticate di specificare quanta malta deve essere messa, e in che misura, costruirete un muro senza di essa. Se dimenticate di indicare l’altezza del muro, il robot continuerà ad impilare mattone su mattone fino a quando per un semplice starnuto tutto cadrà a terra.

Le somiglianze con la programmazione sono strette. Una parola chiave BASIC corrisponde ad un comando come “costruisci un muro” dato all’operaio. Esso eseguirà un mucchio di lavori, basandosi su molte istruzioni che non gli abbiamo dato, ma non potrà lavorare con la rapidità che desiderate. Se, invece, vi accollerete la fatica di dare dettagliatamente ogni minima istruzione al microprocessore (il robot), le eseguirà in maniera incredibilmente veloce. Ma possiamo ampliare l’analogia. Se diceste al vostro operaio di “restaurare il muro”, egli potrebbe non essere in grado di farlo mentre, sotto forma di istruzioni molto dettagliate, il robot potrebbe eseguire questa operazione. Così il linguaggio macchina permette di eseguire delle istruzioni che non sono presenti in BASIC, sebbene sia giusto osservare come sui modelli più recenti di computer il set di istruzioni BASIC sia più esteso che sui precedenti e che questo aspetto del linguaggio macchina non sia più così importante come lo era in passato.

Prima di iniziare a considerare il funzionamento interno del C64 è necessario dare un ultimo sguardo al diagramma a blocchi. Il blocco contrassegnato dal nome “PORT” comprende più di un chip. In linguaggio macchina un PORT è usato per trasmettere delle informazioni, un byte alla volta, all’interno o all’esterno del resto del sistema, la CPU, la ROM e la RAM. La ragione per cui esiste una sezione a parte per trattare i dati è che le entrate (Input) e le uscite (Output) usano operazioni importanti, sebbene lente nell’esecuzione. Usando un PORT permettiamo al microprocessore di scegliere se leggere un dato in entrata o scrivere un dato in uscita. Inoltre si possono separare i dati in entrata e quelli in uscita dalle normali operazioni della CPU. Questa è la ragione per cui

nei programmi in BASIC non appare nulla sullo schermo quando non si fa uso dell'istruzione PRINT. Ed è anche la ragione per cui premendo il tasto PLAY sul registratore, non si ottiene alcun effetto se non tramite l'istruzione LOAD (seguita dal nome del file). I PORT tengono nascoste le operazioni del computer fino a quando non si debba immettere o emettere dati.

Nel gettare uno sguardo a ogni sezione importante del C64, ho fatto liberamente uso dei termini. I puristi contesteranno ad esempio l'uso, da parte mia, della parola "PORT", ma nessuno potrà lamentarsi delle operazioni svolte da esso. Ora passeremo ad analizzare il modo in cui il computer usa la CPU, la ROM, la RAM e i PORT nella programmazione e nell'esecuzione di programmi in BASIC. È giunto il momento di cambiare capitolo.

Capitolo 2

L'interno del Commodore 64

Lontani dal volere aprire il Commodore 64, in questo capitolo vedremo come è progettato per caricare ed eseguire programmi in BASIC. Cominceremo con una versione semplificata del funzionamento dell'intero sistema tralasciando, per il momento, i dettagli.

La ROM del vostro Commodore 64, che inizia all'indirizzo 40960, consiste in un grande numero di piccoli programmi, le subroutine, scritte in linguaggio macchina, assieme alla serie di valori (tavole) uguali alla tavola delle parole chiave. Le parole chiave hanno almeno una subroutine in linguaggio macchina, altre possono richiedere l'impiego di più subroutine. Quando il Commodore 64 viene acceso, la parte di linguaggio macchina che viene eseguita è detta "routine di inizializzazione". Sebbene questo programma sia molto lungo, viene eseguito dal linguaggio macchina al ritmo di molte migliaia al secondo, tanto da non renderne possibile la visione. Tutto ciò che si nota è il tempo che intercorre tra l'accensione e la stampa della nota sui diritti della MICROSOFT. In questo breve tempo, tuttavia, è stato controllato il funzionamento della parte RAM della memoria, una parte di RAM è stata "scritta" con i byte che verranno usati in seguito, mentre la maggior parte della RAM è stata cancellata per potere essere utilizzata.

Il fatto che ne sia stata cancellata una gran parte non significa che nulla sia memorizzato nella RAM. Quando il computer viene spento, la RAM perde ogni traccia dei segnali immagazzinati, ma quando lo si accende di nuovo, le celle di memoria non rimangono tutte azzerate. In ciascun byte, alcuni bit saranno commutati ad 1 mentre altri verranno commutati a 0. Ciò avviene in modo completamente casuale così, se si volesse vedere cosa è presente all'interno di ciascun byte subito dopo aver acceso il computer, si troverebbero dei numeri senza significato. Questi sarebbero sempre compresi tra lo 0 e il 255, la normale serie di numeri per un byte di memoria. Questi numeri sono senza valore, non erano stati messi deliberatamente nella memoria e non costituiscono neppure

istruzioni e dati utilizzabili. Il primo lavoro del computer, quindi, è di azzerare questi valori. Al posto dei numeri casuali, il computer sostituisce una serie molto più ordinata di trentadue byte del valore di 255 seguiti da trentadue byte del valore di 0. Provate a battere (senza numero di linea) quanto segue:

FOR N = 2049 TO 2249:? PEEK (N); " "; :NEXT.

e premete RETURN. La serie di indirizzi di memoria che abbiamo utilizzato costituisce l'inizio del campo del BASIC dove sono normalmente memorizzati i primi byte di un programma in BASIC. Se noi abbiamo appena acceso il computer e non abbiamo usato un numero di linea per il comando in memoria, non dovrebbe esserci nulla eccetto questa serie di numeri. Come potete vedere dallo schermo, la serie consiste principalmente in cifre di 255 e di 0. All'inizio tuttavia, sparsi nella memoria, troverete altri numeri. Questi sono presenti a causa del comando FOR N=2049.. che avete usato per visualizzare la prima serie di numeri, mentre gli altri si trovano lì dal momento in cui è stato eseguito il comando.

Tuttavia, il programma d'inizializzazione compie altre operazioni. La prima sezione di RAM, dall'indirizzo 0 all'818, serve al sistema operativo. Questo perché le subroutine in linguaggio macchina che caricano le funzioni del BASIC hanno bisogno di valori in memoria all'atto della loro esecuzione. I numeri d'indirizzo 43 e 44, ad esempio, contengono l'indirizzo del primo byte di un programa in BASIC. Per immagazzinare numeri per la formazione dei testi e dei grafici che appaiono sullo schermo viene usata una sezione molto più grande di memoria. Inoltre, una parte della RAM serve anche per contenere valori che si creano quando un programma viene eseguito. Ed è quanto andremo a vedere ora.

Le variabili

I programmi in BASIC fanno largo uso di variabili, che utilizzano lettere allo scopo di rappresentare numeri e parole. Ogni volta che "dichiarate una variabile" usando una linea come:

N=20 oppure A$= "SMITH"

il computer deve memorizzare sia il nome della variabile (N oppure A$ o qualsiasi cosa voi usiate), sia il valore (come 20 o SMITH) assegnatole. La parte di memoria usata per ricordare le variabili viene chiamata VLT (tavola elenco variabili). Questa non occupa nessun posto fisso nella memoria, ma è situata nello spazio libero subito dopo il vostro programma. Se si aggiungono una o più linee al programma, l'indirizzo della

VLT deve essere spostato in un settore con un numero d'indirizzo più alto. Se cancellate una o più linee dal vostro programma, la VLT verrà spostata in basso in modo da mantenersi sempre subito dopo l'ultima linea del BASIC.

Ora, poiché la tabella dell'elenco delle variabili può e deve spostarsi non appena si modifica il programma, il computer deve ogni volta annotare il punto in cui inizia la tabella; ciò è possibile mediante l'impiego di due byte nella parte di memoria riservata al sistema operativo, cioè agli indirizzi 45 e 46. Il fatto che vengano usati due indirizzi non deve meravigliarvi. La ragione è che un byte può contenere un numero il cui valore non può superare 255. Usando due byte possiamo scrivere 255 unità in uno di essi, e quello che rimane nell'altro. Un numero come 257, ad esempio, viene scritto come 256 col riporto di uno. Questo numero può essere interpretato come 1,1. Ciò significa che 1 è memorizzato nel byte riservato ai gruppi di 256, ed 1 nel byte riservato alle unità. L'ordine di memorizzazione dei numeri va dal byte basso a quello alto. Per cercare il numero memorizzato bisogna moltiplicare il secondo byte per 256, quindi sommarlo al primo byte. Per esempio, se trovaste 3,8 memorizzato in due indirizzi consecutivi, essi equivarrebbero al numero:

8*256+3=2051

Il numero più alto che possiamo memorizzare utilizzando due byte come questi è 255,255, che significa 255*255+255=65535. Questa è la ragione per cui non è possibile usare cifre molto grandi come 70000 per il numero di linea nel Commodore 64: il sistema operativo usa solo due byte per memorizzare i propri numeri di linea. In realtà, per altre ragioni, il numero più grande utilizzabile è il 63999.

Tutto ciò significa che noi possiamo trovare l'indirizzo memorizzato agli indirizzi 45 e 46 usando la formula:

```
?PEEK(46)*256+PEEK(45)
```

Se la usate subito dopo aver acceso il Commodore 64, otterrete come risultato il numero 2051. Questo si trova subito dopo l'indirizzo nel quale viene memorizzato il primo byte di un programma BASIC. Per vederlo in pratica, battete la linea:

```
10 N=20
```

e provate ancora:

```
?PEEK(46)*256+PEEK(45)
```

Se avrete battuto questa linea con uno spazio tra il numero della linea e la "N", l'indirizzo che otterrete sarà 2060. La tabella dell'elenco delle

variabili si è spostata in avanti di 9 byte. Noterete che questo è un numero di byte più alto di quelli battuti; le ragioni verranno spiegate in seguito.

Molti importanti indirizzi usati dal computer sono "distribuiti dinamicamente" come questo. "Distribuiti dinamicamente" significa che il computer deve cambiare la posizione in cui i byte devono essere mantenuti e che nel contempo terrà traccia di dove sono stati memorizzati, alterando un indirizzo conservato in una coppia di byte come in questo caso. Questo fatto comporta delle implicazioni sul modo di utilizzare il computer. Se, ad esempio, shiftate la VLT cambiando il contenuto agli indirizzi 45 e 46 tramite POKE, il computer non sarà in grado di trovare i valori delle sue variabili. Provatelo, dopo aver trovato l'indirizzo della VLT, ma senza eseguire il programma: 10 N=20, battete ?N.

La risposta sarà zero. La ragione di questo sta nel fatto che non avete eseguito il programma. L'indirizzo 2060 è il punto da cui la VLT parte, ma poiché non vengono create delle VLT fino a quando il programma non viene eseguito, non viene creata nessuna VLT. Per ora sarà più facile per voi cancellare o aggiungere le linee di programma. Tutto quello che dovete variare è il contenuto dei due numeri agli indirizzi 45 e 46. I valori della VLT vengono posizionati solo quando si esegue il programma, ed ogni volta che lo eseguirete verrà creata una nuova tavola. Ogni volta che viene creata una nuova tavola, una coppia di nuovi valori compare all'interno degli indirizzi 45 e 46. Ecco perché dopo avere editato un programma non potete riprendere dal punto in cui l'avevate lasciato, ma dovete eseguirlo di nuovo per creare una nuova VLT ad un nuovo indirizzo.

Se ora eseguite il programma precedente e battete ?N otterrete la risposta che vi aspettate: 20. Ora battete (senza numero di linea) POKE 46,9 e 'RETURN'. Questa operazione permette di cambiare l'indirizzo della VLT in un indirizzo in cui la VLT stessa non è presente. Provate ?N e guardate quello che succede. Sul mio Commodore 64 era un numero molto grande, perché il valore corretto della variabile N non poteva più essere trovato. Se il vostro Commodore 64 si blocca durante questo esercizio, dovrete spegnerlo per poterne riacquistare il controllo. Ciò accade di rado, ma se succede, dovrete spegnere il computer, perdendo il programma in esso contenuto. È da notare, per inciso, l'uso di POKE per cambiare un vecchio valore in un indirizzo della memoria. La forma corretta del comando è POKE A.D. A è un indirizzo che va da 0 a 40959 (il campo di valori della memoria RAM) per il Commodore 64. D è il dato che si vuole registrare all'indirizzo di memoria, e deve essere un valore compreso tra 0 e 255. Se tentaste di usare numeri maggiori di 255 otterreste il messaggio d'errore: '?FC ERROR'.

Uno sguardo alla tavola

È ora tempo di fare qualcosa di più costruttivo e dare un'occhiata al contenuto della VLT. Quando si fanno queste investigazioni, è importante che il computer sia libero dai risultati del precedente lavoro svolto. Perciò è consigliabile spegnerlo e riaccenderlo prima di iniziare qualsiasi altro tentativo. Se premete solo il tasto RESTORE i valori memorizzati in memoria non verranno cambiati. È tedioso, lo so, ma è il vostro linguaggio macchina! Per potere lavorare, dopo aver spento e riacceso il computer, battete la linea:

```
10 N=20
```

di nuovo, e cercate l'indirizzo della VLT usando

```
?PEEK(46)*256+PEEK(45)
```

Esso darà di nuovo l'indirizzo 2060. Ora battete i valori nella VLT, e guardate quello che vi è stato memorizzato. Questa operazione è possibile digitando:

```
FOR X=2060 TO 2070:?X;" "; PEEK(X):NEXT
```

e premendo RETURN. Otterremo la lista illustrata nella figura 2.1. Siamo ora in grado di riconoscere qualcosa? Dovremmo riconoscere il primo byte (78) perché nel codice ASCII corrisponde alla lettera N!.

```
2060    78
2061    0
2062    133
2063    32
2064    0
2065    0
2066    0
2067    88
2068    0
2069    140
2070    1
```

Figura 2.1 - La VLT di immissione per una variabile numerica semplice

Il byte successivo è pari a zero perché la nostra variabile si chiama N e non N1 o NG o qualsiasi altro nome di due lettere. Se usassimo un nome di due lettere, verrebbero occupati entrambi gli indirizzi 2060 e 2061. I successivi cinque byte indicano il modo in cui il numero 20 è stato codificato. A questo punto, non badate a come questi numeri sono usati

per rapppresentare il numero 20: accettatelo come dato di fatto! Come faccio a sapere che sono i 5 byte successivi a rappresentare il numero 20? Facile: il byte all'indirizzo 2067 è 88, che nel codice ASCII rappresenta la lettera X, che è la variabile usata per stampare i valori di una variabile numerica. Il Commodore 64 usa sempre 5 byte per memorizzare qualsiasi valore di una variabile numerica, non importa se questa è data da un piccolo numero come 20, da uno molto più grande come 1427 o 68135, da una frazione o da un numero negativo. Ciò rende la memorizzazione delle variabili più semplice e facilita al computer la loro ricerca.

Se, ad esempio, il computer sta cercando il valore della variabile Y, quando trova la N (78 nel codice ASCII) non perde tempo con i 6 byte successivi (uno per la seconda lettera, 5 per il valore), ma può spostarsi nel punto successivo in cui è memorizzato un nuovo nome. Se siete curiosi, e ci sapete fare in matematica, troverete nell'appendice A il metodo di codificazione usato per convertire i numeri in 5 byte. Gli scopi di questo libro non richiedono la conoscenza del metodo impiegato nella codifica, purché sappiate come viene memorizzato il codice e di quanti byte esso ha bisogno.

La scrittura delle stringhe

Ora dobbiamo guardare come è memorizzata una variabile stringa. Dopo aver spento e riacceso il computer, battete la linea:

10 AB$= "QUESTA È UNA STRINGA".

Eseguite il programma, quindi cercate l'indirizzo della VLT utilizzan-

```
2078    65
2079    194
2080    16
2081    10
2082    8
2083    0
2084    0
2085    88
2086    0
2087    140
2088    2
```

Figura 2.2 - La VLT di immissione per una variabile di stringa

do gli indirizzi 45 e 46 come prima. Io ho ottenuto 2083. Ora usate:

```
FOR X 2078 TO 2088:?X; " "; PEEK(X):NEXT
```

per trovare il contenuto della VLT, mostrato nella figura 2.2. Il primo valore della tavola è 65 che, in codice ASCII, equivale alla immissione nella VLT di una variabile stringa (rappresentante la lettera A). Il secondo è 194. Questo è il valore ASCII della B e il computer capisce che si tratta di una variabile stringa. Se aveste usato il nome A$ per la variabile al posto di AB$, il secondo numero memorizzato (all'indirizzo 2679) sarebbe stato 128, e non 0. Quando utilizzate una variabile numerica, il secondo codice ASCII del nome sarà 0, oppure uno dei numeri del codice ASCII, mai più grande di 127.

Ora diamo uno sguardo al resto. Non assomiglia molto al codice ASCII, non è vero? Infatti le entrate consistono di solo sette byte, la stessa lunghezza di una variabile numerica. Si può capire quello che accade solo dando uno sguardo ai numeri. Il numero che segue il codice di B (194, poiché 128 è stato aggiunto al codice ASCII) è 16. Esso fornisce il numero dei caratteri che formano la stringa. Se contate il numero delle lettere e degli spazi vedrete che le cose stanno così. I seguenti due byte sono 10 e 8. Poiché due byte assieme assomigliano a un indirizzo, combinandoli assieme nel modo solito (8*256+10) otterremo 2058. Il passo successivo è PEEK (2058) con il risultato di 84, il codice ASCII di T. 2058 allora è l'indirizzo del primo byte della stringa.

Riassumendo: il Commodore 64 memorizza una immissione di 7 byte nella propria VLT per ogni stringa. Di questi, i primi due sono riservati per il nome della stringa. Il secondo varrà 128 o più. Quando si usa un nome di due caratteri, al secondo codice viene aggiunto il valore 128, il che permette al computer di distinguere una variabile stringa da una variabile numerica. I 5 byte successivi contengono la lunghezza della stringa e l'indirizzo nella memoria del primo suo byte. Infatti, sono necessari solo tre byte per rintracciare una stringa. Un byte è necessario per la lunghezza, nessuna stringa può superare la lunghezza di 255 caratteri (in realtà non è possibile immettere più di 240 caratteri). Due byte vengono impiegati per l'indirizzo, così che due dei sette byte utilizzati nell'immissione della stringa VLT sono necessari solo come separatori. La convenienza di avere la stessa lunghezza complessiva nell'immissione della VLT sia per una stringa, sia per un numero, mette in ombra la piccola perdita degli ultimi due byte in ogni immissione di stringa.

In questo esempio, la stringa è memorizzata ad un indirizzo minore della VLT, nella parte della memoria dei programmi BASIC e i codici ASCII per la stringa vengono messi qui quando battete il programma. I numeri devono essere trasferiti nella VLT poiché non sono memoriz-

zati come codici ASCII. Ora la domanda che ci si pone è: "cosa accade quando si crea una stringa non presente all'interno del programma?". Spegnete e riaccendete il computer, e battete:

10 A$="AB":B$="CD":C$=A$+B$.

Eseguendo il programma noterete che la vostra VLT è più lunga. Questa volta dovrete guardare gli indirizzi della memoria a partire dalla locazione 2080 fino a quella 2100. Troverete le immissioni delle stringhe A$ e B$, le quali forniscono gli indirizzi all'interno dell'area della memoria del programma, come potete vedere nella figura 2.3. La variabile C$, tuttavia, fornisce i byte 252, 159 per il suo indirizzo. Ciò corrisponde all'indirizzo 40956 (cioè 159*256+151, ricordate!) per questa stringa. Possiamo dare uno sguardo a questi indirizzi. Se battete:

FORX=40956TO40959:?X;" ";PEEK(X);" ";CHR$(PEEK(X)):NEXT

potrete vedere ogni cosa. Ora i codici ASCII delle lettere ABCD sono immagazzinati qui, e l'uso di CHR$ permette di poterli vedere.

```
2080    65
2081    128
2082    2
2083    9
2084    8
2085    0
2086    0
2087    66
2088    128
2089    2
2090    17
2091    8
2092    0
2093    0
2094    67
2095    128
2096    4
2097    252
2098    159
2099    0
2100    0
```

Figura 2.3 - L'immissione della VLT per una stringa che non è memorizzata nell'area di programma della memoria

I numeri interi

Abbiamo visto come vengono memorizzati i numeri e le stringhe, ma non dobbiamo dimenticare che il Commodore 64 permette di memorizzare due tipi di numeri. Uno di questi viene chiamato "reale" mentre l'altro viene chiamato "intero". Una variabile per un numero reale usa una lettera o un paio di lettere, o una lettera e una cifra (ad es.: A, AB o A2) per rappresentarlo. Un numero reale viene memorizzato nel modo che abbiamo già visto, usando in totale 7 byte. Di questi, due sono per il nome della variabile, e 5 per il suo valore. Un numero reale può essere negativo, positivo o frazionario e il suo valore può variare da 10^{38} circa a 10^{-39}. I numeri interi non ammettono parti frazionarie e possono andare da —32768 a +32767. Ora prendiamo visione di questi numeri nella VLT.

Iniziamo, come al solito, spegnendo e poi riaccendendo il computer per azzerare la memoria. Ora battete la linea:

```
10 A%=15: B%=300
```

e fatela eseguire al computer. Trovate la posizione della VLT come al solito usando l'istruzione PEEK agli indirizzi 45 e 46. Nel mio Commodore 64 si trova alla locazione 2068. Ora battete

```
FOR X=2068TO2085: PRINT X;" ";PEEK (X): NEXT
```

e premete RETURN. Ciò vi restituisce la sequenza mostrata in figura 2.4.

```
2068   193
2069   128
2070   0
2071   15
2072   0
2073   0
2074   0
2075   194
2076   128
2077   1
2078   44
2079   0
2080   0
2081   0
2082   88
2083   0
2084   140
```

Figura 2.4 - L'immissione della VLT per due numeri interi

Non è come la sequenza dei numeri che abbiamo visto nella fig. 2.1. Per iniziare bisogna fare qualcosa nel primo byte, quello che dovrebbe rappresentare la variabile dal nome A%. Il byte è il 193, che è la somma di 65+128. Esso rappresenta il codice della A+128. Il secondo byte è 128. Ciò suggerisce che quando si crea un nome di variabile intera, la macchina aggiunge 128 ad entrambe le lettere del nome. Riuscite a vedere le caratteristiche di tutto questo? Per un numero reale, le lettere dei nomi usano i codici ASCII. Per una stringa il secondo codice del nome è 128, o un codice ASCII aggiunto a 128. Per un numero intero, il numero 128 è stato aggiunto al primo codice allo stesso modo. Ecco come la macchina distingue due variabili a lettere diverse. I simboli $ e % non vengono registrati nella memoria!

Ora però dobbiamo dare un'occhiata al modo in cui i numeri vengono memorizzati. I byte che seguono i byte del nome della variabile sono 0, 15, 0, 0, 0, e ciò sembra indicare che sia stato memorizzato il numero 15. Questo procedimento non assomiglia molto alle trasformazioni che abbiamo visto durante la memorizzazione di un numero reale. Il numero 300, tuttavia, viene memorizzato come 1,44. Potrebbe essere un sistema di memorizzazione a due byte? Verifichiamolo: 256*1+44=300, il numero è stato memorizzato usando due byte, partendo dal byte più alto, diversamente da quanto accade per l'ordine dei numeri di linea o degli indirizzi della memoria. L'immissione della VLT è ancora di 7 byte, ma con tre di essi non utilizzati.

Come si spiega l'uso dei numeri interi? In primo luogo si può spiegare il campo di estensione dei numeri interi. Se usiamo solo due byte per la memorizzazione, non si può memorizzare un numero superiore a quello che utilizza 255 immagazzinato in ogni byte. Questo numero sarebbe 255*256+255, che dà 65536. Ora sappiamo che la gamma dei numeri interi possibili va da —32768 a +32767 che sommati danno 65535. Invece ricoprire la gamma di numeri interi da 0 a 65535, il Commodore 64 comprende una gamma più utile a partire da —32768 fino a +32767. Le ragioni della scelta di questa gamma di numeri è spiegata più dettagliatamente nell'appendice A. L'uso dei numeri interi comporta altri effetti collaterali. Uno di questi è che i numeri interi vengono memorizzati con una perfetta precisione. Quando dite che A%=17412, questo sarà il numero memorizzato, e non 17411.9999999. Un numero reale è quasi sempre approssimato, e usando numeri reali, dobbiamo essere pronti ad un arrotondamento. Avrete probabilmente incontrato calcolatori che vi forniscono la risposta 3.999999 invece di 4.0. Ciò è causato da queste approssimazioni nella memorizzazione dei numeri, mentre certi calcolatori arrotonderebbero il numero a 4 ed altri non lo farebbero. Se utilizzerete i numeri interi sul Commodore 64, non avrete tali problemi. L'al-

tro vantaggio nell'uso degli interi è la rapidità. Poiché il computer tratta due byte invece di cinque (numeri reali), è più rapido avendo una decodifica più semplice. Se preferite la velocità fate uso dei numeri interi.

I programmi

È il momento di dare uno sguardo a come un programma viene memorizzato nella memoria del Commodore 64. Come prima, dovremo avvalerci del comando PEEK per vedere cosa sta succedendo in certe zone della memoria. La prima cosa che dobbiamo sapere, comunque, è dove si trovano i byte che formano gli indirizzi di inizio programma. Essi sono memorizzati nei byte 43 e 44.

```
10 A=10
20 PRINT A
30 C$="CBM 64"
```

Figura 2.5 - Un semplice programma in BASIC

Possiamo perciò iniziare osservando come un programma è presente nella memoria. Digitate il programma come è mostrato nella figura 2.5, ma non eseguitelo. Ora digitate:

?PEEK(44)*256+PEEK(43)

e troverete l'indirizzo in cui inizia il primo byte di questo programma. In questo esempio, il Commodore 64 fornisce l'indirizzo 2049. Usando il solito ciclo per stampare i valori dei numeri tramite PEEK da questo indirizzo in poi, otterrete la lista mostrata nella figura 2.6.

10	8	10	0	65	178	49	48	0	18	8
20	0	153	32	65	0	34	8	30	0	67
36	178	34	67	77	66	32	54	52	34	0

Figura 2.6 - I byte che rappresentano il programma in memoria

A prima vista essa appare come una serie senza senso di numeri, ma ad una analisi più attenta, vi scorgerete certe caratteristiche. Come al solito i codici ASCII fungono da efficaci segnali. All'indirizzo 2053, ad esempio, potrete vedere il numero 65, che è il codice ASCII di 'A'. Siccome sappiamo che la linea è "A=10", possiamo cercarne il resto. Il dieci è riconoscibile come 49 (ASCII '1') e 48 (ASCII '0'), perciò il numero 178 deve rappresentare il carattere operazionale '='. Ma questo non è il codice ASCII che sta per '=', bensì uno di quei 'token' che ho menzionato nel capitolo 1. Si tratta di un token perché al computer è richiesto di eseguire un'operazione e non solo di memorizzare un codice ASCII in questo punto. Lo 0 all'indirizzo 2057 segna la fine di questa linea.

Ora dobbiamo affrontare i primi quattro byte. I primi due sono, come potete immaginare guardandoli, un indirizzo. I numeri 10 e 8 formano il numero 8*256+10, cioè 2058. Che indirizzo è questo? È l'indirizzo del primo byte della linea successiva! Ecco come il sistema operativo del Commodore 64 scova le linee di programma e le pone nella sequenza corretta, non importa quale ordine usiate nell'immetterle. Il mistero finale è finalmente risolto. Guardando il terzo e quarto byte di ciascuna linea appare la sequenza 10,20,30— i numeri di linea. Esistono due byte riservati per i numeri di linea perché si ha bisogno di linee superiori a 255. Per i numeri di linea minori di 256, il secondo di questi byte — il più significativo — non viene utilizzato.

Ora diamo uno sguardo alle altre linee, come appaiono immagazzinate nelle memoria. Abbiamo incontrato il token PRINT 153 di prima, e tutto il resto dovrebbe essere familiare. La sola novità è la fine del programma. L'ultima linea termina con uno 0, come al solito, ma dopo di esso, nel punto in cui dovrebbero esserci i due byte della linea successiva, c'è un altro paio di zeri. Questo è il marcatore di fine programma usato dal computer (END).

Si possono apportare alcuni interessanti cambiamenti ad un programma come questo. Supponiamo, ad esempio, di andare a cambiare il contenuto degli indirizzi (tramite POKE) usati per contenere il numero di linea. Se digitate:

```
POKE2060,10:POKE2068,10
```

e premete RETURN, avrete scritto il numero 10 in ogni indirizzo del numero di linea per le linee 20 e 30. Ora listate e guardate i risultati! È un programma formato da sole linee '10'. Contrariamente a quanto potreste aspettarvi, esso sarà eseguito perfettamente allo stesso modo di prima. La possibilità di eseguire un programma dipende, come potete capire, dall'indirizzo della linea di programma corretta successivamente, e non da come vengono numerate le linee. Un programma variato

in questo modo certamente non è normale. Provate, ad esempio, ad usare l'edit di schermo per cambiare la seconda linea A in B, in modo tale da leggere PRINT B. Ora eseguite il programma, quindi listatelo. Scoprirete che la prima linea precedente è scomparsa, e che la nuova prima linea è PRINT A, con PRINT B come seconda linea! Potete, tuttavia, registrare su cassetta un programma che è stato alterato come questo, e ricaricarlo normalmente. Questo è il metodo in embrione col quale si può rendere difficile l'alterazione di un programma.

Funzionamento del programma

Dopo aver analizzato il modo in cui un programma viene codificato e memorizzato nella memoria del Commodore 64, possiamo dare uno sguardo a come esegue le istruzioni. Questa operazione viene eseguita dalla parte più complicata del sistema operativo, e necessita dell'indirizzo di partenza. Esso viene memorizzato nelle locazioni numero 43 e 44, che abbiamo già usato. Supponiamo di portare a termine le operazioni, tralasciando i dettagli, delle tre linee del programma della figura 2.5. Al primo indirizzo nel BASIC, la subroutine RUN leggerebbe i primi due byte e li memorizzerebbe temporaneamente. Questi due byte verrebbero usati al posto dell'indirizzo di inizio BASIC quando la linea successiva viene eseguita. I numeri di linea vengono quindi letti e memorizzati. Perché? Perché se si presentasse un errore di sintassi nella linea, il computer sarebbe in grado di inviare un messaggio di errore: "SYNTAX ERROR IN 10" cioè "ERRORE DI SINTASSI ALLA LINEA 10" e non solo "SYNTAX ERROR SOMEWHERE" cioè "ERRORE DI SINTASSI DA QUALCHE PARTE". Il byte successivo è un codice ASCII, e il computer lo tratterà come nome di variabile. In passato, le variabili dovevano essere dichiarate usando la parola chiave LET, token ancora in uso sui moderni computer. La differenza è che ora il token è immesso automaticamente quando un numero di linea o un due punti è seguito da una lettera. Allo stesso modo, se digitate LET, il computer utilizzerà lo stesso token.

Seguendo il token di assegnamento, il token del simbolo "=" fa sì che entri in azione la subroutine di quella operazione. Questa crea una immissione nella tavola della lista delle variabili, nel primo indirizzo disponibile, e pone il codice ASCII di A in quel punto. L'indirizzo successivo nella VLT viene lasciato vuoto poiché il nome della variabile è formato da una sola lettera. Quindi viene letto il numero 10 e convertito in forma binaria, come appare nell'appendice A. Anche questo gruppo di byte viene messo nella VLT come l'immissione di A. Viene quindi letto il byte successivo nel programma che, essendo 0, fa sì che si passi al-

l'indirizzo della linea successiva del programma che è stata letta come prima operazione, e che viene memorizzata all'interno del microprocessore. Il tipo di operazione che abbiamo visto in dettaglio per la linea 10 si ripete allora per la linea 20. Questa volta, dovranno essere fatte molte più cose quando verrà letto il token dell'istruzione. Dato che questo è il token di PRINT, dovrà essere richiamata la subroutine PRINT. Essa individuerà l'indirizzo del successivo posto vuoto dello schermo. Ciò è possibile usando una coppia di byte della RAM per tenere nota dell'indirizzo — leggete questi byte e otterrete l'indirizzo. Quindi viene trovato il valore di A nella VLT, e i byte convertiti di nuovo nella forma del codice ASCII. I codici vengono posti, uno per uno, nella memoria dello schermo. Facendo ciò, si fa sì che i caratteri diventino visibili sullo schermo, tramite un'altra subroutine. Di nuovo, lo zero posto al termine della linea fa sì che il computer passi a una linea nuova di programma. Alla fine della terza linea, tuttavia, il numero della linea successiva è zero, e il programma finisce. Il computer ritorna allo stato comandi, in attesa di un nuovo comando.

Non così semplice come può sembrare dalla descrizione, ma sostanzialmente è ciò che accade. La cosa importante da comprendere è che ci sono varie operazioni da svolgere, le quali devono essere fatte una alla volta, passo per passo. Ciò che rende lento il BASIC è che ogni token richiama una subroutine, la quale deve essere trovata. Ad esempio, se avete un programma che consiste in un ciclo come:

```
10 FOR N=1TO50
20 PRINT N
30 NEXT
```

l'operazione di lettura del token PRINT di 153 è cercare il luogo in cui l'esatta subroutine viene memorizzata e che verrà eseguita 50 volte. Non esistono metodi semplici per assicurarsi che la subroutine sia localizzata una volta e quindi utilizzata 50 volte. Il tipo di BASIC che avete sul Commodore 64 è un INTERPRETE BASIC. Ciò significa che ogni istruzione è calcolata come se il computer andasse da sé. Questo vuol dire cercare la subroutine di PRINT 50 volte. L'alternativa è uno schema chiamato COMPILAZIONE, nel quale l'intero programma viene tradotto in linguaggio macchina prima di essere eseguito. Per compilare un programma, si utilizza un altro programma chiamato COMPILATORE. L'uso di un compilatore per un programma BASIC rende l'esecuzione del programma molto più veloce, ma rende anche la sua correzione più difficile, poiché trasforma il programma in un'altra forma. Non si può avere sempre tutto!

Capitolo 3

Il microprocessore

In questo capitolo, inizieremo a familiarizzare col microprocessore 6502 del Commodore 64. Il microprocessore o CPU è, se vi ricordate, la parte del computer che, a differenza della parte che memorizza i dati o di quella destinata all'immissione o all'uscita dei dati (port), "agisce" in modo tale da tenere sotto controllo ciò che fa tutto il resto del computer.

La stessa CPU consiste di una serie di sezioni di memoria per i numeri, ma con una organizzazione in aggiunta. Per mezzo dei circuiti che così propriamente vengono chiamati "cancelli", il modo in cui i byte vengono trasferiti all'interno delle diverse parti della memoria del CPU può essere controllato: sono queste operazioni a formare l'addizione, la sottrazione, le operazioni logiche ed altre, della CPU. Nulla accade a meno che un byte di istruzione non si presenti nella forma di segnali 0 e 1 ad ognuno degli otto terminali (piedini) dei dati, e questi byte sono usati per controllare i cancelli all'interno della CPU. Ciò che rende l'intero sistema così utile è che, poiché le istruzioni del programma sono sotto forma di segnali elettrici su otto linee, questi segnali possono essere cambiati molto rapidamente. La velocità viene decisa da un altro circuito elettrico chiamato "generatore di impulsi clock", o "clock" in breve. La velocità scelta come standard per il clock del Commodore 64 è assai elevata, così da potere eseguire un milione di operazioni al secondo.

Il linguaggio macchina

Il programma per la CPU, come abbiamo visto, consiste di codici numerici, con ogni numero compreso tra lo 0 e il 255 (un numero dà un solo byte). Alcuni di questi numeri possono essere byte di istruzioni che fanno sì che la CPU esegua qualcosa. Altri potrebbero essere byte di dati, i quali sono numeri da aggiungere, o immagazzinare o spostare, o che

potrebbero essere codici ASCII per le lettere. La CPU non è in grado di dare delle motivazioni, ma funziona semplicemente nel modo in cui le è stato detto di funzionare. Sta al programmatore sistemare i numeri nella sequenza corretta.

L'ordine corretto, per quanto riguarda la CPU, è molto semplice. Il primo byte che viene trasmesso dopo aver acceso il computer o dopo aver terminato una istruzione, viene considerato come byte di istruzione. Ora, molte delle istruzioni del 6502 consistono di un solo byte e non hanno bisogno di dati. Altre possono essere seguite da uno o due byte di dati, e qualche istruzione ha bisogno di due byte. Quando la CPU legge un byte di istruzioni, analizza l'istruzione che stabilisce se è una di quelle che devono essere seguite da uno o più byte. Se, ad esempio, il byte d'istruzione è un byte che deve essere seguito da due byte di dati, allora quando la CPU analizza il primo byte tratterrà i due byte successivi che gli vengono inviati come byte di dati per l'istruzione. Questa operazione della CPU è completamente automatica, ed è presente all'interno della CPU stessa. La difficoltà è che il programmatore in linguaggio macchina deve lavorare seguendo le stesse regole, ed ottenere un programma corretto. Se trasmettete al microprocessore un byte di istruzione quando esso si aspetta un byte di dati, o viceversa, allora vi troverete nei guai. Per guai si intende, quasi sempre, un ciclo senza fine, che svuota lo schermo azzerandolo e priva i tasti delle loro funzioni. Perfino la combinazione dei tasti (RUN STOP) (RESTORE) può fallire quando si desidera riportare il Commodore 64 fuori da tale ciclo, e il solo rimedio è quello di spegnerlo. In generale, in casi come questo, si perde qualsiasi programma in memoria, perciò è di vitale importanza salvarli, siano essi programmi in linguaggio macchina o in linguaggio BASIC, in modo che le operazioni in linguaggio macchina (attraverso l'uso di POKE) vadano su nastro prima di usare qualsiasi programma.

Ciò che voglio sottolineare a questo punto è che la programmazione in linguaggio macchina è tediosa. Comunque non è necessariamente difficile — si tratta di preparare delle semplici istruzioni per una semplice macchina — l'unica cosa a volte difficile è ricordarsi tutti i particolari richiesti. Quando si programma in BASIC, i messaggi di errore che appaiono sullo schermo aiutano a non commetterne o a identificarli se ci sono. Facendo uso del linguaggio macchina, vi trovate da soli e dovete risolvere i vostri errori. A questo proposito, il tipo di programma chiamato assembler aiuta considerevolmente. Rivedremo questo punto più avanti. Nel frattempo, il miglior modo di apprendere qualcosa circa il linguaggio macchina è di scriverlo, farne uso e compiere i propri errori.

Iniziamo quindi analizzando i metodi per scrivere i numeri che costituiscono i byte del programma in linguaggio macchina.

La notazione binaria, decimale, esadecimale

Un programma in linguaggio macchina consiste in un set di codici numerici. Dato che ogni codice numerico è un modo di rappresentare gli 1 e gli 0 all'interno di un byte, esso consiste di numeri compresi tra lo 0 e il 255 quando lo scriviamo nella notazione decimale. Il programma non può essere di alcuna utilità fino a quando non viene trasmesso alla memoria del Commodore 64, poiché la CPU è un dispositivo veloce e il solo modo di inviarle i byte alla velocità con cui li utilizza è di memorizzarli nella memoria (centrale) e di lasciare che la CPU li usi prendendoli secondo il loro ordine. Non vi sarà possibile digitare numeri così velocemente da tener testa alla CPU, e i metodi del nastro o del disco non sono sufficientemente rapidi.

L'immissione di byte all'interno della memoria, quindi, è parte essenziale di un reale funzionamento di un programma in linguaggio macchina; vedremo poi i metodi nei dettagli minimi. Allo stesso tempo, alcuni programmi semplici e molto corti possono essere messi in memoria usando il metodo piú semplice, tramite 8 interruttori. Ogni interruttore può essere predisposto per dare un output elettrico 0 o 1, e un pulsante può essere premuto per fare sì che la memoria numerizzi il numero che gli interruttori rappresentano, per poi selezionare il successivo indirizzo della memoria. Chiaramente questi programmi sono piuttosto noiosi. Dato che abbiamo a disposizione un computer, mi sembra logico utilizzarlo per inserire numeri in memoria usando una numerazione più conveniente.

La scelta della numerazione conveniente dipende dal modo in cui impostate i numeri e dall'uso che fate della programmazione in linguaggio macchina. Il C64 contiene delle subroutine che convertono i numeri binari nella sua memoria in numeri decimali da stampare sullo schermo e viceversa. Quando usate PEEK, l'indirizzo che desiderate può essere scritto in decimali e il risultato dell'istruzione PEEK sarà un numero decimale compreso tra 0 e 255. Usando POKE potete digitare sia il numero di indirizzo sia il byte da inserire nel decimale.

Comunque, i programmatori in linguaggio macchina giudicano l'uso dei decimali tutt'altro che conveniente. Un numero decimale per un byte può essere una cifra (come 4) o due (come 17) o tre (come 143). Un codice molto più conveniente è quello chiamato esa (abbreviazione di esadecimale). Tutti i numeri di un byte possono venir rappresentati da due numeri semplici esa. In relazione a questo, chi progamma seriamente in linguaggio macchina scrive il programma in quello che viene chiamato linguaggio Assemblatore (ASSEMBLY). Quest'ultimo usa parole per i comandi che sono rappresentazioni abbreviate dei nomi dei comandi della CPU. Quindi, i programmi chiamati Assemblatori convertono queste

parole di comando nei corretti codici binari. In pratica tutti gli assemblatori mostrano questi codici sullo schermo nella forma esadecimale invece di quella decimale. Per di più, quando battete i numeri dei dati, dovrete ricorrere al sistema esadecimale. "Esadecimale" significa in base 16, e la ragione per cui è così largamente usato è che esso è particolarmente adatto per rappresentare i contenuti in sistema binario dei byte. Quattro bit, la metà di un byte, rappresentano numeri che vanno da 0 a 15 nel nostro normale sistema decimale. Questa è la gamma delle cifre esadecimali (vedere fig. 3.1). Siccome non esistono cifre maggiori di 9, dobbiamo fare uso delle lettere A,B,C,D,E ed F in aggiunta ai numeri da 0 a 9 nel sistema esadecimale.

Il vantaggio consiste nel fatto che si può rappresentare un byte tramite un numero di due sole cifre, e un indirizzo completo con un numero di 4 cifre. I codici numerici usati come istruzioni sono stati progettati nel sistema esadecimale, in modo che si possano scorgere meglio le relazioni tra i comandi. Ad esempio, potremmo scoprire che un set di comandi collegati tra di loro iniziano con la stessa cifra se scritti nella forma esadecimale. Nella forma decimale questo particolare non apparirebbe. Inoltre, usando la forma esadecimale è molto più facile scrivere i numeri binari che il computer usa nella realtà. L'uso dell'assembler del Commodore 64 e dei programmi supervisori, come il MIKRO, richiedono una certa familiarità col sistema esadecimale, e i libri che parlano del 6502 saranno tutti scritti partendo dal presupposto che voi conosciate il sistema esadecimale. Sembra sia venuto il momeno di iniziare a parlarne.

Esa	Decimale	Esa		Decimale
0	0	C		12
1	1	D		13
2	2	E		14
3	3	F		15
4	4		poi	
5	5	10		16
6	6	11		17
7	7		a	
8	8	20		32
9	9	21		33
A	10	22		34
B	11		ecc.	

Fig. 3.1 - Cifre decimali ed esadecimali

La notazione esadecimale

Il sistema esadecimale utilizza 16 cifre che normalmente partono dallo 0 per procedere fino al 9. Tuttavia, il numero successivo non è 10 poiché ciò significherebbe $1\times16^1+0\times16^0=16$. Siccome non si hanno a disposizione cifre superiori al 9, si utilizzano le lettere dell'alfabeto dalla A alla F. Il numero 10 del sistema decimale viene scritto 0A nel sistema esadecimale, 11 viene scritto 0B, 12 diventa 0C e così via fino a 15, cioè 0F. Sebbene non sia strettamente necessario farlo, i programmatori si abituano a scrivere i byte di dati usando due cifre e gli indirizzi usando 4 cifre anche se si tratta di numeri piccoli. Il numero che segue 0F è 10, 16 nel sistema decimale; lo stesso discorso si ripete fino a 1F (31 dec.), il numero più grande all'interno di un byte è 255 (dec.), cioè FF (4 hex.). Quando si scrivono numeri nel sistema esadecimale, si è soliti contrassegnarli con un particolare simbolo, per non confonderli con quelli in base 10. Sebbene non ci sia alcuna possibilità di scambiare un numero come 3E con un numero del sistema decimale, i problemi sorgono quando se ne incontra uno come 26. La convenzione usata dai programmatori del microprocessore 6502 per contrassegnare un numero nel sistema esadecimale è il simbolo del dollaro"$" ponendolo prima del numero. Ad esempio, il numero$47 significa 47 esadecimale, ma il solo numero 47 significherebbe il numero quarantasette nel sistema decimale. Quando si scrivono numeri esadecimali in un programma per il microprocessore 6502, è consigliabile seguire la convenzione riassunta in fig. 3.2.

Esa	Binario	Esa	Binario
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Figura 3.2 - Cifre esadecimali e cifre binarie

Il grande vantaggio del sistema esadecimale è la sua stretta corrispondenza al sistema binario. Se date uno sguardo alla figura 3.2, potrete vedere come $9 corrisponda a 1001 (in binario) e $F a 1111. Perciò il numero $9F corrisponde al numero binario 10011111. Non dovete fare altro che trascrivere i numeri binari corrispondenti alle cifre esadecimali. La conversione nella direzione opposta è altrettanto semplice: basta raggruppare le cifre del numero binario in gruppi di 4, partendo dalla cifra

Conversione: esadecimale → binario

Es.: $2C .... $2 = 0010 (bin.)
$C = 1100 (bin.)

perciò $2C è uguale a 00101100 (bin.) e rappresenta un byte di dati.

Es.: $4A7F .... $4 = 0100 (bin).
$A = 1010 (bin.)
$7 = 0111 (bin.)
$F = 1111 (bin.)

perciò $4A7F è uguale a 0100101001111111 (bin.) e rappresenta un indirizzo.

Conversione: binario → esadecimale

Es.: 01101011 .... 0110 = $6
1011 = $B

perciò 01101011 è uguale a $6B

Es.: 1011010010010 (il numero di byte è incompleto)

Disponete a gruppi di 4, partendo dal primo bit di destra:
0010 = $2
1001 = $9
1101 = $D
e le rimanenti 10 = 2, cioè $2D92

Figura 3.3 - Conversione fra i sistemi binario ed esadecimale

meno significativa (quella all'estrema destra nel numero), quindi trasformare ogni gruppo nella cifra esadecimale corrispondente. La figura 3.3 mostra alcuni esempi della conversione in entrambe le direzioni, facendovi capire quanto facile essa sia.

Il Commodore 64 non possiede programmi residenti per la conversione dei numeri esadecimali e binari; il mezzo più conveniente per farlo è un programma in BASIC. La figura 3.4 mostra un programma di conversione dal sistema binario a quello esadecimale che costruisce una stringa chiamata H$ come numero esadecimale. La conversione inizia dividendo il numero per 4096 e considerando la sola parte intera del risulta-

to. Se il numero in base dieci è minore di 4096 il risultato è 0, che rappresenta anche la prima cifra esadecimale. La conversione del sistema esadecimale viene fatta utilizzando i valori dei codici ASCII, poiché c'è una stretta relazione tra i codici ASCII dei numeri da 0 a 9 e le cifre stesse.

```
5 PRINT"♥"
10 PRINT"SCRIVETE UN NUMERO DECIMALE ":INPUT D
20 IFD>65535THENPRINT"LIMITE 65535":GOSUB300:GOTO5
30 IFD<1THENPRINT"NON MINORE DI 1":GOSUB300:GOTO5
40 IFD<>INT(D)THENPRINT"SOLO INTERI":GOSUB300:GOTO5
50 F=4096:H$=""
60 Y=INT(D/F)
70 GOSUB 200
80 D=D-Y*F:F=INT(F/16)
90 IF F<1 THEN 110
100 GOTO 60
110 IF LEFT$(H$,2)="00"THEN H$=RIGHT$(H$,2)
120 H$="$"+H$
130 PRINT"IL NUMERO ESADECIMALE E' ";H$
140 END
200 IF Y<=9THEN H$=H$+CHR$(Y+48)
210 IF Y>9THEN H$=H$+CHR$(Y+55)
220 RETURN
300 FORN=1TO1000:NEXT:RETURN
```

Figura 3.4 - Programma per la conversione dei numeri decimali in esadecimali

Se si aggiunge 48 al numero, si ottiene il suo codice ASCII, perciò $48+1=49$ è il codice di "1". Questo codice ASCII può essere incorporato all'interno di una stringa. Quando si raggiunge il 10 sorge un piccolo problema, poiché esso in esadecimale è rappresentato dalla lettera A, e il suo codice ASCII è 65, che supera il 10 di 55 unità. Il programma di conversione deve aggiungere 48 ai numeri fino al 9 compreso, e aggiungere 55 ai numeri dal 10 al 15 per effettuare l'esatta conversione. Questo non è difficile a farsi in un programma BASIC. Il passo successivo consiste nel prendere il resto della divisione per 4096 e di dividerlo per 256, cioè 4096/16. La parte intera del risultato verrà messa nella stringa sotto forma esadecimale. Il processo si ripete fino a quando non rimane più nulla da dividere. Se volete rendere superefficiente questa routine, usate variabili intere con ogni numero.

```
5 PRINT"♥"
10 Y=1:D=0:PRINT"NUMERO ESADECIMALE":INPUTH$
30 L=LEN(H$):FORN=0TOL-1
40 GOSUB 200:NEXT
50 PRINT"IL NUMERO DECIMALE E' ";D
100 END
200 P$=MID$(H$,L-N,1):A=ASC(P$)
210 IFA<48 OR A>102 THEN GOSUB 300:GOTO 280
220 IF A<65 AND A>57 THEN GOSUB 300:GOTO 280
230 IF A<=97 AND A>70 THEN GOSUB 300:GOTO 280
240 IF A<=57 THEN Q=R-48
250 IF A>=65 THEN Q=A-55
260 IF A>=97 THEN Q=A-87
270 D=D+Q*Y:Y=Y*16
280 RETURN
300 PRINT"ESADECIMALE ERRATO...RIPROVATE":END
500 FORX=1TO1000:NEXT:RETURN
```

Figura 3.5 - Programma per la conversione dai numeri esadecimali in numeri decimali

La conversione dall'esadecimale al decimale può essere fatta in gran parte allo stesso modo; un programma adatto è illustrato in figura 3.5. Esso si basa sulla conversione del codice ASCII di ogni cifra del numero esadecimale nel numero in base 10 corrispondente, da moltiplicare poi per il corrispondente fattore posizionale (1,16,256,4096 fino a 4 cifre).

I numeri ottenuti in questo modo vengono sommati in un totale "D". Il programma in Basic è così semplice che tutte le riviste lo pubblicano con compenso agli autori per ogni nuovo computer apparso sul mercato.

Se volete effettuare queste conversioni quando non avete un C64 a disposizione, leggete l'appendice B.

Supponendo, a ragione, che non vogliate sobbarcarvi il costo di un assemblatore completo, cosa potete fare per scrivere programmi in linguaggio macchina? La risposta è di preparare il vostro programma in Assembler, che è il modo più semplice per scrivere programmi in linguaggio macchina, e di convertirli poi nel sistema esadecimale. Convertire significa cercare il numero esadecimale che rappresenta ogni istruzione nelle tavole, chiamate "set di istruzioni", che vengono messe a disposizione dai produttori di qualsiasi microprocessore; la Mostek, che ha progettato il 6502, ne fornisce uno per questo chip. Per aiutarvi è stata inserita una guida di riferimento in questo stesso libro, nell'appendice C, ma per il momento non usatela perché vi porterebbe fuori strada!

Numeri negativi

Programmi come questi, per la conversione decimale-esadecimale, sono utili per il C64. Però, sotto un certo aspetto, mancano di qualcosa: non sono in grado di trattare i numeri negativi. Questo è un fattore problematico anche se comprensibile. I numeri negativi rivestono una grande importanza all'interno dei programmi in linguaggio macchina, in particolar modo se non si lavora con un programma assemblatore. La ragione sta nel fatto che talvolta si richiede alle CPU di compiere l'equivalente in BASIC di un GOTO, ad esempio saltare al passo che si trova 30 passi in avanti dall'indirizzo attuale. Queste cose vengono di solito programmate fornendo dei dati numerici che rappresentano il numero dei passi che si desidera saltare. Se si vuole tornare indietro ad un passo precedente, tuttavia, si dovrà usare un numero negativo per il byte di dati. Ciò accade spesso poiché è il modo in cui un ciclo è programmato in linguaggio macchina.

Perciò abbiamo bisogno di sapere come si scrive un numero negativo in esadecimale. Ciò che rende scomoda questa operazione è che non ci sono segni negativi nell'aritmetica esadecimale. Allo stesso modo non ne esistono nel sistema binario.

La conversione di un numero nella sua forma negativa viene effettuata usando un metodo chiamato della complementazione e la figura 3.6 mostra il procedimento. A prima vista, e molto spesso anche dopo, ciò sembra del tutto senza senso. Quando si sta trattando un numero di un solo byte, ad esempio, la forma binaria per —1 è 255! Si usano numeri molto grandi per rappresentarne uno piccolo negativo! Si inizia a dare un senso alle cose quando si guardano i numeri nella notazione binaria. I numeri che possono essere considerati negativi iniziano con un 1, mentre quelli positivi iniziano con uno 0. Il CPU riesce a distinguerli andando a controllare il bit più significativo (quello all'estrema sinistra del byte). È un metodo semplice, che la macchina riesce ad usare efficientemente, ma che comporta degli svantaggi per le persone. Uno di questi svantaggi è che le cifre di un numero negativo non sono le stesse di quelle di un numero positivo. Ad esempio, —40 (dec.) usa le stesse cifre di +40 (dec.). In esadecimale —40 diventa $D8 e +40 $28. Il numero decimale —85 diventa $AB e + 85 diventa $55. Il secondo svantaggio è che una persona non è in grado di distinguere tra byte il cui numero è negativo e byte il cui numero è maggiore di 127. Ad esempio, $9F significa 159 o —97? La risposta a questa domanda è che l'operatore non se ne deve preoccupare. Il microprocessore userà il numero correttamente, al di là di quello che noi possiamo pensare. Il problema è che noi dobbiamo sapere qual è l'uso corretto in ogni singolo caso.

Binario	Numero (8 bit)	00110101 (53 dec.)
Passo 1	cambia ogni 0 in 1, ogni 1 in 0	11001010
Passo 2	Aggiungi 1	+1
		11001001 (203 dec.)

Il risultato è la forma negativa di un numero.

Decimale	Numero	53
Passo 1	Sottrarre da 256	203

Questa è la forma negativa (in base 10)

Esadecimale	Numero	$35
Passo 1	Sottrarre da $FF	FF
		35
		$CA

Ricordare che F rappresenta 15 (dec.) e 15—5=10 (dec.) che è A (esadec.).

Passo 2	Aggiungere 1	+1
		$CB

Questa operazione è più semplice di una sottrazione con $100 che comporta l'uso del riporto in esadecimale. Il risultato è la forma esadecimale del numero negativo. Un'altra strada percorribile è la conversione in decimale, per poi riconvertire il risultato in esadecimale.

Figura 3.6 - Il complemento a 2, o forma negativa, di un numero binario

In questo, come in altri libri che parlano della programmazione in linguaggio macchina, vedrete usare parole come "segnato" e "senza segno". Un numero segnato può essere negativo oppure positivo. Per un numero di un byte, i valori che vanno da 0 a $7F sono positivi, mentre i valori che vanno da $80 a $FF sono negativi. Questi corrispondono ai numeri decimali da 0 a 127 per i valori positivi e da 128 a 255 per quelli negativi. Se vedete il numero $9C definito come "segnato", allora saprete che viene trattato come numero negativo (poiché è superiore a $80). Se il numero viene definito senza segno, allora esso è positivo, ed il suo valore viene calcolato per semplice conversione.

Come si converte un numero esadecimale di un singolo byte segnato in un numero decimale, usando il nostro programma? È semplice. Se il numero è maggiore di $7F, allora sottraete 256 dal suo valore decimale. Se, ad esempio, si ottiene 240 come risultato, allora 240—256=—16, che è il valore segnato nel sistema decimale.

Un diversivo

Per staccarci un attimo da tutta questa aritmetica e fare una pausa, diamo uno sguardo alla configurazione dello schermo del Commodore 64. Ogni sua parte può essere controllata da tutto ciò che è presente in una parte della memoria, ma esistono due tipi di memoria. La più semplice con cui lavorare è quella chiamata memoria dello schermo. Essa occupa gli indirizzi della memoria a partire dalla locazione $400 fino a $7E7, cioè da 1024 a 2023 in decimale. Ciò che si indica con "memoria dello schermo" è una parte della memoria trattata in maniera speciale. Ogni cosa che viene qui memorizzata, viene poi usata per visualizzare qualcosa sullo schermo. Ciò significa che ogni codice numerico che viene memorizzato a un indirizzo riprodurrà la lettera, il numero o il carattere grafico corrispondente nel suo punto di influenza sullo schermo. Non è però così semplice sul C64. Quando un numero viene posto in un punto di questa memoria, i suoi effetti non sono visibili a meno che non si verifichi una delle sue seguenti condizioni.

La prima è che deve essere già presente un carattere in quel punto dello schermo. La seconda è che si deve usare un nuovo colore per lo sfondo. I lettori della mia guida al BASIC del Commodore 64, COMMODORE 64 COMPUTING, saranno già al corrente di queste condizioni. Provate quanto segue: premete il tasto CLEAR per azzerare lo schermo, quindi digitate:

```
POKE53281,3:POKE1524, 1
```

e premete il tasto (RETURN). Vedrete apparire la lettera A nel centro dello schermo. Il computer ha convertito il suo codice "interno" 1, memorizzato nella posizione 1524, in una serie di numeri che produrranno la lettera "A" sullo schermo in quel punto. Per programmare questo tipo di cose si può ricorrere ad un ciclo, come è mostrato nella figura 3.7. Per prima cosa ripristinate le condizioni normali dello schermo premendo (RUN/STOP) e (RESTORE).

```
10 POKE 53281,3
20 FORN=1024 TO 2023
30 POKEN,1:NEXT
```

Figura 3.7 - Un programma per riempire lo schermo di lettere "A"

Ciò che si ottiene da questo ciclo è il riempimento dello schermo di lettere A. Il riempimento non è particolarmente veloce, perché stiamo

usando un ciclo del BASIC. Più avanti, faremo lo stesso discorso, però in linguaggio macchina, e vedrete che è sbalorditivamente più veloce! Per il momento, però, guardate cosa accade se si va a sostituire il codice 1 con un altro codice appartenente ai blocchi grafici, come 65. È molto utile, e se si vuole evitare di vedere comparire la scritta READY seguita dal cursore lampeggiante, aggiungete un ciclo senza fine come segue:

40 GOTO 40.

La cosa strana qui è l'uso di codici "interni". Il C64 usa i numeri del set di codici standard ASCII nel suo BASIC. Quando usate delle istruzioni BASIC come ASC e CHR$, state usando dei codici numerici ASCII. Per i suoi scopi, comunque, il C64 converte questi codici ASCII in altri numeri. Questi sono i codici numerici "interni", che sono elencati nell'Appendice E del manuale del C64. Quando immettiamo un numero nella sezione di memoria riservata allo schermo, il carattere che apparirà sarà scelto dall'elenco "interno", e non dai codici ASCII come potreste aspettarvi.

Capitolo 4

La struttura del microprocessore 6502

I registri

PC e accumulatore

Un microprocessore è formato da un insieme di memorie, chiamate registri. Queste memorie sono di tipo abbastanza diverso se paragonate alla ROM e alla RAM. I registri sono collegati l'un l'altro ai piedini del corpo della CPU da circuiti chiamati cancelli (gate). In questo capitolo, daremo uno sguardo ai registri di maggiore importanza del 6502 e come essi vengono usati. Un buon punto di partenza è il registro chiamato PC, abbreviazione di Program Counter.

Questo registro non conta i programmi, ma conta i passi di un programma. Il PC è un registro di due byte in grado di memorizzare fino a 65535 ($FFFF) numeri di indirizzi. Il suo scopo è quello di contare i numeri degli indirizzi. Il numero memorizzato nel Program Counter viene incrementato di una unità per ogni istruzione portata a termine, o quando è necessario un altro byte. Ad esempio, se il PC contiene l'indirizzo $1F3A (7994 dec.), e in questo indirizzo è contenuta un'istruzione, allora il PC lo incrementerà a 7995 ($1F 3B esad.) quando la CPU sarà pronta per accogliere un nuovo byte. Il byte successivo sarà letto da questo nuovo indirizzo.

Ciò che rende il PC così importante è il modo automatico di fare uso della memoria. Quando il PC contiene un indirizzo, i segnali elettrici che corrispondono agli 0 e agli 1 di quell'indirizzo compaiono su un gruppo di connessione, chiamati nell'insieme BUS degli indirizzi, i quali collegano la CPU con l'intera memoria, RAM e ROM. Il numero contenuto nel PC seleziona un byte dalla memoria: quello contenuto a quell'indirizzo. All'inizio di un'operazione di lettura, la CPU invierà un segnale, chiamato segnale di lettura, su un'altra linea; ciò farà sì che la memoria colleghi la parte selezionata con un gruppo di linee, il BUS dei dati.

I segnali sul BUS dei dati avranno le stesse caratteristiche dei segnali (0 e 1) che compaiono nel byte della memoria selezionato dall'indirizzo contenuto nel PC. Ogni volta che il numero all'interno del PC cambia, viene selezionato un nuovo byte dalla memoria, in modo tale che la CPU possa sempre avere a disposizione dei byte. Quando la CPU è pronta per un nuovo byte, allora il PC viene incrementato e viene emesso un nuovo segnale di lettura.

Vi sono altri modi in cui può essere cambiato il Program Counter, ma per il momento li tralasciamo per considerare un altro registro: l'ACCUMULATORE.

L'accumulatore di un microprocessore è il registro della CPU "che agisce". Ciò significa che ne farete normalmente uso per immagazzinare qualsiasi numero che volete trasferire da qualsiasi altra parte, o al quale aggiungere, o su cui compiere qualsiasi altra operazione. Il nome "Accumulatore" deriva dal modo in cui questo registro opera. Se si ha un numero memorizzato nell'accumulatore e vi si aggiunge un altro numero, allora anche il risultato viene memorizzato in esso. L'equivalente più prossimo in BASIC è lo scrivere, usando la variabile A, questa linea:

```
A=A+N
```

dove N è una variabile numerica. Il risultato di questa linea BASIC è quello di aggiungere ad A il valore N e di assegnare il nuovo valore ancora alla variabile A. L'accumulatore agisce nello stesso modo, con la sola differenza che esso non è in grado di accogliere numeri superiori a 255 (dec.).

Il 6502 ha un registro accumulatore, spesso contrassegnato come "REGISTRO A". L'importanza di questo è che viene usato molto di più dell'altro, poiché molte azioni possono essere portate a termine più velocemente, più facilmente o, forse, solo nell'accumulatore. Quando un byte viene letto dalla memoria, di solito lo si pone nell'accumulatore. Quando si calcola una qualsiasi operazione logica o aritmetica, essa viene normalmente eseguita all'interno dell'accumulatore così come il risultato che ne consegue.

Metodi di indirizzamento

Quando si programma in BASIC non ci si deve affatto preoccupare degli indirizzi della memoria a meno che non si faccia uso delle istruzioni PEEK e POKE. Il compito della ricerca di un byte è svolto dal sistema operativo del computer. Quando si assegna un valore ad una va-

riabile in un programma BASIC come, ad esempio, nella linea:

10 N=12

non dobbiamo mai preoccuparci da dove o in che forma venga memorizzato il numero 12. Allo stesso modo, quando si aggiunge la linea:

20 K=N

non dobbiamo minimamente preoccuparci dove il valore N era stato messo e dove verrà messo il valore K. Se ricordate il nostro esempio della costruzione del muro, programmando in linguaggio macchina dobbiamo specificare ogni numero di cui faremo uso, o altrimenti l'indirizzo in cui questo numero è stato memorizzato. Il metodo usato per ottenere questi numeri, o trovare un luogo in cui metterli, viene chiamato metodo di indirizzamento. Ciò che rende particolarmente importante la scelta dei metodi di indirizzamento è che per ognuno di essi il codice numerico delle istruzioni è diverso. Ciò significa che la stessa istruzione esiste in più di una versione, con un codice differente per ogni versione. La lista di tutti i metodi di indirizzamento del 6502, a questo punto, creerebbe confusione, e per questa ragione è stata messa nell'appendice D. Ora invece analizzeremo alcuni esempi di metodi d'indirizzamento e il modo in cui si scrivono nel linguaggio ASSEMBLER.

Il linguaggio Assembler

Tentare di scrivere un programma direttamente in linguaggio macchina è molto difficile e passibile di errori dall'inizio alla fine. Il metodo più usato nell'iniziare a scrivere un programma consiste nello scriverlo in una serie di passi in quello che viene chiamato linguaggio ASSEMBLATORE (o linguaggio ASSEMBLER). Questo linguaggio è formato da un set di parole abbreviate che formano delle istruzioni, chiamate "mnemoniche", e numeri che formano i dati o gli indirizzi. I numeri possono essere in notazione esadecimale o decimale, a patto che vengano forniti al computer nella forma corretta. Ogni linea di un programma in linguaggio Assembler indica un'operazione del microprocessore; più tardi questo set di istruzioni viene "assemblato" in linguaggio macchina, da cui questo nome.

Lo scopo di ogni linea di un programma in linguaggio macchina è di mostrare le operazioni e i dati o gli indirizzi necessari per eseguire quella operazione. Allo stesso modo in cui noi facciamo uso dell'istruzione TAB in BASIC, abbiamo bisogno di completare il comando con un numero. La parte del linguaggio Assembler che specifica ciò che deve esse-

re fatto viene chiamata OPERATORE, mentre quella che specifica su cosa l'operazione è eseguita, viene chiamata OPERANDO. Sono poche le istruzioni che non hanno bisogno dell'operando, e di queste ci occuperemo in parte più avanti.

Facciamo un esempio prendendo in considerazione la seguente linea di programma in Assembler:

```
LDA #$12
```

L'operatore è LDA, la forma abbreviata di LOAD A; ciò significa che si deve caricare un byte nel registro accumulatore A. L'operando è #$12 di cui $12 esadecimale, invece di 12 decimale L'altro simbolo #, castelletto, viene usato per indicare il metodo di indirizzamento che deve essere usato, chiamato "indirizzamento immediato".

Quindi l'intera linea dovrebbe sortire l'effetto di porre il numero $12 nel registro accumulatore A. Ciò equivale a dire, in un programma BASIC

A=18 (ricordate che $12 equivale a 18 dec.).

Capirete che la memoria che conteneva il numero è all'interno del microprocessore piuttosto che nella memoria RAM, e che è contrassegnata col nome "A".

Il comando LDA #$12 utilizza l'indirizzamento immediato, poiché il byte caricato all'interno dell'accumulatore deve essere posto nella locazione della memoria il cui indirizzo segue immediatamente quello del byte dell'istruzione. È come lasciare un bigliettino al vostro lattaio, su cui è scritto: "Il denaro si trova in una busta ai piedi della porta accanto". Esiste un codice numerico per la parte LDA# dell'intera istruzione. Questo byte è $A9, così la sequenza in notazione esadecimale A9 12 rappresenta l'intero comando LDA#$12. Essendo tuttavia molto più facile ricordare il significato dell'istruzione LDA# $12 invece che A9 12, è meglio usare il più possibile il linguaggio Assembler.

L'indirizzamento immediato può essere conveniente, ma impegna nell'uso di un dato numero e di un indirizzo della memoria ben preciso. È come dire in BASIC

N=4*12+3

invece di

N=A*B+C.

Nel primo esempio, N non può essere nient'altro che 51, e non avremmo potuto scrivere che N=51. Il secondo esempio è più elastico, e il valore di N dipende dai valori scelti per le variabili A,B e C. Quando nella memoria RAM si trova un programma in linguaggio macchina, allora

i numeri che vengono caricati da questo metodo di indirizzamento immediato possono essere cambiati se necessario. Tuttavia, se tale programma è contenuto nella ROM, non è possibile effettuare nessun cambiamento. Ecco perché sono necessari altri metodi di indirizzamento. Uno di questi è l'"indirizzamento assoluto".

L'indirizzamento assoluto utilizza un indirizzo formato da due byte come operando. Ciò crea un mucchio di lavoro per il 6502 e questo perché, dopo avere letto il codice per l'operatore, deve leggere altri due byte per trovare l'indirizzo in cui si trovano i dati. Esso dovrà mettere nel PC questo indirizzo, letto nel byte dei dati, eseguire l'operazione e quindi riporre il corretto byte successivo nel PC. La figura 4.1 mostra sotto forma di diagramma ciò che si deve fare. Un'operazione di indirizzamento assoluto è perciò molto più lenta nell'essere eseguita di una immediata, ma siccome qualsiasi byte può essere memorizzato all'indirizzo specificato, è facile cambiarne i dati.

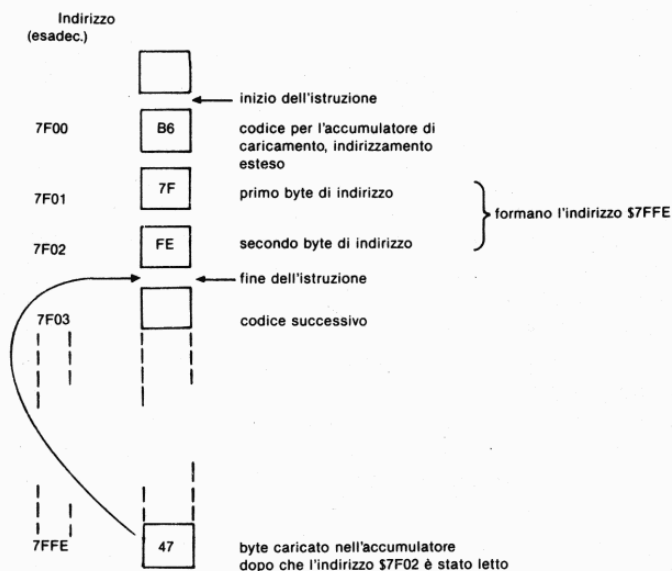



Figura 4.1 - Il funzionamento del metodo di indirizzamento assoluto

Supponiamo, ad esempio, di avere l'istruzione:

LDA # $7FFE

In questa parte di linguaggio macchina, l'operatore è LDA (carica l'accumulatore A), e l'operando è l'indirizzo $7FFE. Ciò che dovrete ricordare è che quello che viene messo all'interno del registro A non è 7FFE, che è l'indirizzo di due byte, ma il byte di dati che viene memorizzato nella memoria a questo indirizzo. Lo scopo di tutta l'istruzione è, quindi, di copiare il byte memorizzato alla locazione $7FFE nell'accumulatore A del 6502. Quando l'istruzione è stata completata, l'indirizzo $7FFE contiene ancora la propria copia del byte, poiché la lettura della memoria non altera in alcun modo il contenuto.

L'indirizzamento assoluto può essere usato anche in un comando che memorizzi un byte all'interno della memoria. Il comando

STA $7FFF

significa che il byte memorizzato nell'accumulatore A deve essere copiato in memoria all'indirizzo $7FFF. In questo modo si modifica il contenuto di questo indirizzo di memoria, ma l'Accumulatore A contiene lo stesso byte anche dopo l'esecuzione di un'istruzione.

Indirizzamento a pagina zero

L'indirizzamento a pagina zero è un metodo che permette di specificare un intero indirizzo usando un solo byte! Il segreto sta nel fatto che il byte superiore dell'indirizzo viene considerato pari a zero. Ecco da dove deriva il nome "pagina zero" per questo tipo di indirizzamento. Supponiamo, ad esempio, di aver usato il comando:

LDA $3F

Qui non compare nessun simbolo cancelletto (#), perciò non significa che il numero $3F venga caricato nell'accumulatore. Ciò che succede è che si carica l'accumulatore del byte memorizzato nella locazione $003F. I due 00 rappresenta la pagina zero, mentre 3F rappresenta la parte specificata nell'istruzione. La gamma di indirizzi utilizzabili con questo metodo va dalla locazione $0000 alla locazione $00FF; cioè solo 256 (dec.) byte, ma molto importanti. Nel microprocessore 6502, l'indirizzamento a pagina 0 ci permette di accedere rapidamente a qualsiasi indirizzo fra questi, utilizzando un solo byte di indirizzo (il byte basso). Ciò rende questi indirizzi i preferiti da qualsiasi programmatore che desidera eseguire caricamenti e memorizzazioni rapide. Per questa ragione, gli indirizzi "pagina 0" della RAM in qualsiasi computer che monti il 6502 sono i migliori per una memorizzazione di grande quantità. Sappiamo che vengono usati per memorizzare quantità come l'indirizzo di inizio del

BASIC e l'indirizzo della tavola dell'elenco delle variabili (VLT). Gli stessi indirizzi vengono usati per qualsiasi tipo di quantità importanti, e perciò nella trattazione di questo libro ne analizzeremo molti con maggior dettaglio. Poiché il C64 fa uso di molti indirizzi di "pagina zero" per i suoi scopi, si deve fare attenzione a come li usiamo nei nostri programmi. Se tentassimo di usare un indirizzo di cui il computer ha bisogno, bloccheremmo l'intero sistema. Questa è una ragione per cui è meglio "salvare" un programma in linguaggio macchina prima di provarlo.

Indirizzamento indicizzato

L'indirizzamento indicizzato è un metodo particolarmente utile sul 6502. Il principio è che un registro ad 8 bit è usato per contenere un byte. Questo byte viene quindi aggiunto a un indirizzo base quando viene usato l'indirizzamento indicizzato. Ad esempio, supponiamo di avere all'interno di un registro indice il numero $4C. Se si specifica poi che vogliamo caricare dall'indirizzo $7000 con indicizzazione, allora il numero nel registro indice verrà aggiunto all'indirizzo $7000. L'indirizzo diventa così $704C, e questo è il numero che verrà usato come numero della locazione. Il risultato è che il byte nella locazione $704C verrà caricato nell'accumulatore.

Nel 6502 sono presenti due registri che possono essere usati in questo modo: i registri X e Y. Sono entrambi registri di otto bit e sono quasi identici, ma hanno alcune differenze che risulteranno più importanti in seguito. Una delle differenze che possiamo prendere subito in considerazione è l'indirizzamento indicizzato a pagina zero. Quando carichiamo il registro con un byte come $4A, e quindi formiamo il comando (in linguaggio Assembler):

```
LDA $7000,X,
```

significa che l'indirizzo che verrà usato è $7000 più il contenuto del registro X. Il risultato è che viene usato l'indirizzo $704A. Questo è l'indirizzamento indicizzato assoluto. "Assoluto" significa che si sta usando un indirizzamento assoluto per "l'indirizzo base" di $7000, e si aggiunge il numero "indice" di 4A dal registro indice X. Possiamo usare il registro indice Y del 6502 nello stesso esatto modo, con istruzioni come:

```
LDA $7000,Y,
```

in linguaggio Assembler. L'uso del registro indice X, tuttavia, ci permette di utilizzare l'indirizzamento a pagina zero. Possiamo, ad esem-

pio, usare il comando in linguaggio Assembler come:

LDA 0C, X

che significa che l'indirizzo base è $000C, e il numero contenuto nel registro X viene aggiunto a $000C prima che venga usato. Se il numero contenuto nel registro X è $23, allora $000C + $23 dà $002F che è il numero dell'indirizzo che verrà usato. L'accumulatore verrà perciò caricato dall'indirizzo $002F. L'indirizzamento indicizzato a pagina zero non può essere usato col registro indice Y, così che il comando comune:

LDA $1F, Y

è impossibile: non esistono codici di istruzione per esso.

L'uso dell'indirizzamento indicizzato potrebbe non sembrare particolarmente utile a prima vista. Tutto quello che state facendo, dopo tutto, è di aggiungere un numero ad un indirizzo. Ciò che rende questo metodo così utile è che si può cambiare il numero contenuto nei registri X o Y. Più in particolare, si può incrementare o decrementare il numero in ambedue i registri. Il comando INX significa "incrementa X". L'effetto è di aggiungere 1 al numero memorizzato nel registro X. Ora, siccome il numero nel registro X viene aggiunto all'"indirizzo base" quando si usa l'indirizzamento indicizzato, incrementare X significa che incrementeremo gli indirizzi che si ottengono da una operazione di caricamento o di memorizzazione indicizzata di X.

Supponiamo, ad esempio, che vogliate memorizzare 10 byte in 10 locazioni contigue. È un problema molto comune, come ad esempio la stampa di 10 caratteri sullo schermo. L'uso dell'indicizzazione significa che si può attivare un ciclo che memorizzi, usando l'indicizzazione, per poi incrementare l'indice e ripetere l'operazione di memorizzazione. Eseguite l'operazione per un totale di 10 volte, così da completarla.

Forse è un po' presto per fare menzione dell'uso di un ciclo, ma lo incontreremo presto. L'esempio è utile, poiché esso illustra uno degli usi più comuni dell'indirizzamento indicizzato. Oltre a incrementare i registri X e Y (INX, INY), possiamo anche decrementarli. Il comando in linguaggio Assembler DEX decrementa X (sottrae 1 dal numero contenuto in X), mentre DEY decrementa Y. Siccome abbiamo due registri indice, è anche possibile caricare da una locazione X base indicizzata, quindi incrementare X. Il byte caricato può ora essere memorizzato, indicizzato nel registro Y questa volta usando un altro indirizzo base. Il registro Y può quindi essere decrementato. Se tutto questo viene fatto in un ciclo, si avrà che i byte verranno spostati da un gruppo di indirizzi di partenza in un altro gruppo di indirizzi, fino a un altro indirizzo di partenza. Complicato, vero? Potrebbe sembrarlo a parole, ma nei fatti

è un metodo semplicissimo e chiaro per spostare i byte da una parte della memoria all'altra: operazione molto spesso essenziale nell'elaborazione dei dati.

Indirizzamento indiretto

L'indirizzamento indiretto significa andare ad un indirizzamento per alleggerirne un altro in cui si trova il byte richiesto. È come andare all'indirizzo di un hotel. Il 6502 permette di usare due metodi principali di indirizzamento. Questi sono relativamente complicati e di essi non faremo grande uso in questo libro, perché abbiamo voluto scrivere una introduzione, non una enciclopedia. Possiamo tuttavia dare uno sguardo ai principi chiamati in causa, perché i due metodi indiretti si assomigliano sotto molti punti di vista.

Il principio più importante è ricorrere all'uso degli indirizzi a pagina zero appaiati. In ciascuna di queste coppie si può immagazzinare un indirizzo a due byte. L'ordine dei byte che dovrete per ora conoscere è il byte basso e quello alto. Il metodo di indirizzamento indiretto fa uso del primo paio di questi indirizzi a pagina zero. L'effetto di un comando che fa uso dell'indirizzamento indiretto è perciò leggere il byte nel primo degli indirizzi a pagina zero, e metterlo nella parte bassa del registro PC. Viene quindi letto il successivo indirizzo a pagina zero, e il byte contenuto in esso viene messo nella parte alta del Program Counter. Il Program Counter ora contiene un indirizzo completo che viene usato per caricare o memorizzare, a seconda dell'operazione richiesta. Ad esempio, se l'indirizzo \$10 contiene \$3D e l'indirizzo \$11 contiene \$7F, l'effetto di un caricamento indiretto da \$10 sarebbe di mettere la locazione \$7F3D nel PC, e così l'accumulatore si carica col byte memorizzato all'indirizzo \$7F3D. Di nuovo, sembra molto complicato e non necessario fino a quando non comprenderete gli speciali vantaggi di tale metodo. Il vantaggio speciale è che si può cambiare l'indirizzo usato alterando i numeri memorizzati nella memoria di pagina zero.

Il sistema operativo del C64 fa grande uso dell'indirizzamento indiretto con gli indirizzi memorizzati nella pagina zero della RAM. I metodi di indirizzamento indiretto del 6502 sono, infatti, più complicati di quanto non abbia qui scritto, perché vengono utilizzati anche i registri indice. Uno dei metodi di indirizzamento indiretto è illustrato negli esempi dei Capitoli 7 e 9.

Indirizzamento relativo

L'indirizzamento relativo è stato uno tra i primi metodi di indirizzamento usati, ma oggi è utilizzato in pochi comandi.

L'indirizzamento relativo significa che l'operando di un'istruzione può essere formato da 1 o 2 byte, e l'indirizzo che sta per essere usato si trova aggiungendo questo numero (chiamato "offset") all'indirizzo "corrente", che è il numero del Program Counter. È simile alle mappe vecchio stile dell'Isola del Tesoro che specificano un passo a sinistra, due in avanti, tre a destra... eccetera. Non sapete dove andrete a finire se non conoscete il punto da cui iniziare, ma quando si usa l'indirizzamento relativo in un microprocessore, il punto di partenza è di solito quello dell'indirizzo contenuto nel PC. Il 6502 usa l'indirizzamento relativo solo per i comandi di salto, e l'OFFSET è un byte trattato come un numero segnato. L'uso di un numero segnato di un solo byte significa che si può saltare a un nuovo indirizzo che si trovi 127 passi in avanti o 128 passi indietro dall'indirizzo attuale. Questi salti sono gli equivalenti in linguaggio macchina del GOTO, ma con la differenza che possono essere fatti dipendere da una condizione, come l'accumulatore contenente zero. È come se ci fosse un'unica istruzione BASIC che sortisce l'effetto di:

IF A=0 THEN GOTO...

Analizzeremo le istruzioni di salto più avanti.

Gli altri registri

Il registro 5 è un registro di 8 byte che per il momento tralasceremo. Questo registro viene chiamato "STACK POINTER" (puntatore di pila), e viene usato per individuare i byte che la CPU ha memorizzato temporaneamente. Se si interferisce con ciò che è stato memorizzato nel registro S, si sconvolge il sistema operativo del computer.

L'altro registro importante per noi è il Processor Status (P) Register che analizzeremo ora nei dettagli.

Il registro P

Il Processor Status Register, a volte chiamato il FLAG REGISTER, non è in realtà un registro come gli altri.

Non si può fare nulla con i bit in questo registro; essi non sono neanche associati come numero. Lo Status Register viene usato come tastiera elettronica. Sette bit del registro (in tutto ce ne sono 8) vengono usati

per registrare quanto è accaduto nella fase precedente del programma. Se il passo precedente era una sottrazione che lascia il contenuto del registro A a zero, uno dei bit dello Status Register passerà dal valore 0 a 1 per avvertire la CPU di questo. Se si aggiunge un numero a quello dell'accumulatore, e il risultato consiste di 9 bit invece di 8 (fig. 4.2), allora un altro bit dello Status Register assume il valore 1. Se il bit più significativo di un registro passa da 0 a 1 (che potrebbe voler dire un numero negativo), allora viene settato (posto pari a 1) un altro bit dello Status Register. Ogni bit, quindi, viene usato per sapere cosa è appena accaduto. Ciò che rende importante questo registro è che si possono avere istruzioni di salto a seconda del valore di un bit dello Status Register.

Numero dell'accumulatore	10110110
Numero aggiunto	11000101
Risultato	101111011

Questo consiste di 9 bit, mentre l'accumulatore ne può contenere solo 8. Il bit più significativo viene trasferito al FLAG CARRY dello Status Register. Ora l'accumulatore contiene 01111011. Il byt CARRY (di riporto) è settato (=1).

Figura 4.2 - La ragione per cui il byt CARRY è necessario

La figura 4.3 mostra come i bit dello Status Register del 6502 vengono stabiliti. Di questi bit, 0,1 e 7 sono quelli più probabilmente usati all'inizio della programmazione. L'uso degli altri è più specifico di quanto per ora abbiamo bisogno. Il bit 0 è il bit CARRY (o FLAG CARRY).

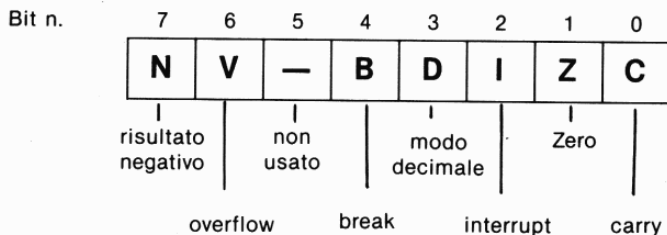



Figura 4.3 - I bit del Processor Status Register. Solo 3 di questi, N, Z, C, sono molto usati nella maggior parte dei programmi

Questo viene settato (a 1) se un riporto dal bit più significativo è risultato non essere nullo. Quando si sta eseguendo una sottrazione (o una operazione simile come il confronto), allora questo bit verrà usato per indicare se è stato necessario andare in "prestito". Per alcuni scopi può venire utilizzato come nono bit dell'accumulatore, particolarmente per operazioni di spostamento e rotazione in cui i bit di un byte vengono tutti spostati di una posizione (fig. 4.4).

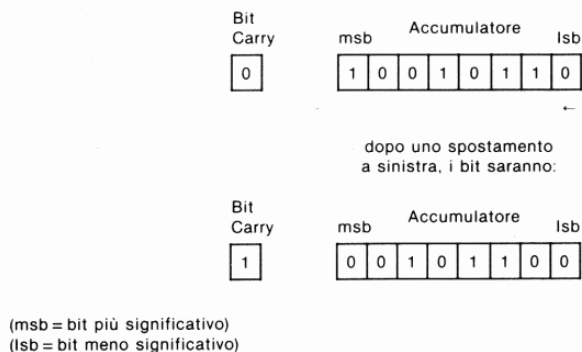



Figura 4.4 - Uso del Bit Carry (Flag Carry) in una operazione di spostamento, in cui ogni bit di un byte viene spostato a sinistra di una posizione

Il Flag Zero è il bit 1. È settato se il risultato dell'operazione precedente è stato precisamente zero, altrimenti viene azzerato (reset=0). È un utile mezzo per stabilire l'uguaglianza fra due byte — sottrarre 1 all'altro — e se il Flag Zero è settato, allora i due byte erano uguali. Il Flag negativo viene settato se il numero che risulta in un registro dopo un'operazione ha il suo bit più significativo uguale a 1. Questo è il tipo di numero che potrebbe essere negativo se stessimo lavorando con numeri segnati; questo bit, perciò, viene largamente impiegato quando si opera su tali numeri.

Alcuni bit dello Status Register possono essere variati in maniera selettiva, ma non è certo pane per i denti dei principianti! Di solito non si carica o copia nulla in questo registro. Esso viene usato quasi esclusivamente come segnalatore di quello che è successo e, limitatamente a questo, ne verrà illustrato l'impiego. Altri espedienti possono aspettare fino a quando non siate diventati esperti.

Capitolo 5

Le funzioni dei registri

Operazioni dell'accumulatore

Dato che l'accumulatore è il principale registro ad un solo byte, dobbiamo ora elencarne in dettaglio le funzioni esadecimali.

Di tutte le operazioni dell'accumulatore, il semplice trasferimento di un byte è di gran lunga il più importante. Non possiamo ad esempio ricavare nessuna forma aritmetica con i numeri del codice ASCII, e così le operazioni principali che eseguiremo su questi byte saranno il caricamento e la copiatura. L'accumulatore viene caricato di un byte copiato da una locazione di memoria, e memorizzato in un'altra. Pochissimi sistemi di computer permettono di spostare un byte direttamente da una locazione all'altra, così che viene usato quasi esclusivamente uno strano metodo per farlo.

Il successivo gruppo di funzioni più importanti è il gruppo aritmetico-logico il quale contiene la somma, la sottrazione, AND e OR. A questo gruppo si possono aggiungere le funzioni per lo spostamento e la rotazione che abbiamo preso brevemente in considerazione nel capitolo precedente. Gli effetti delle istruzioni di spostamento e rotazione, con le relative mnemoniche in linguaggio Assembler è mostrato nella figura 5.1. Uno spostamento risulta sempre in un registro privato di uno dei bit che in esso sono contenuti e cioè quello all'estremità indicata dal verso dello spostamento. Entrambi i tipi di spostamento fanno sì che nel registro si aggiunga uno zero all'estremo opposto del byte. Il bit carry viene usato come nono bit dell'accumulatore in entrambi questi spostamenti. L'operazione di spostamento può essere eseguita nel registro A (l'accumulatore) oppure in un byte residente in memoria. L'effetto di uno spostamento su un numero binario memorizzato nel registro è di moltiplicare il numero per due se lo spostamento è verso sinistra, dividerlo per due se lo spostamento avviene verso destra (figura 5.2). La rotazione, al contrario, mantiene inalterati gli stessi bit contenuti nel registro, cambiandone però la posizione. Il 6502 possiede due comandi per la rotazione:

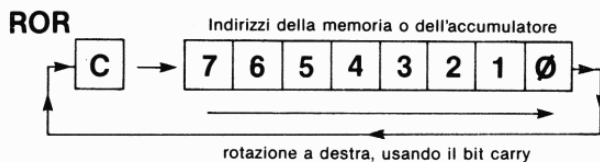
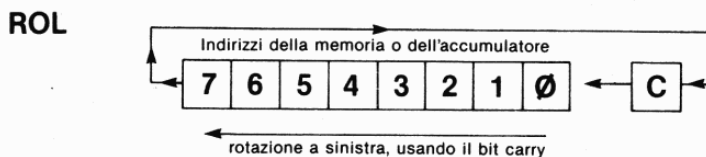
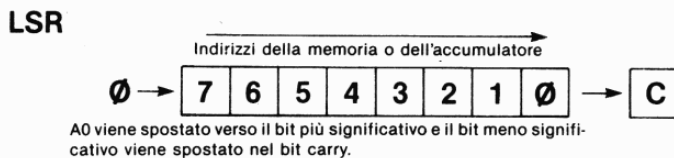
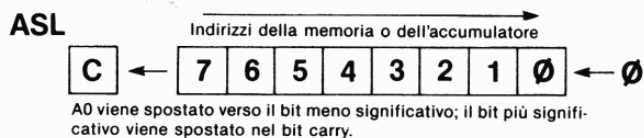



Figura 5.1 - Le istruzioni di spostamento e di rotazione del 6502: ASL (Arithmetic shift left = spostamento a sinistra aritmetico), LSR (Logic shift right = spostamento logico a destra), ROL (Rotate left = rotazione a sinistra), ROR (Rotate right = rotazione a destra)

Figura 5.2 - L'effetto dello spostamento su un numero

uno per la rotazione a sinistra ed uno per quella a destra. Ancora una volta questi usano il bit carry come nono bit del registro. Può essere usato l'accumulatore, oppure l'operazione può essere eseguita da un byte residente in memoria.

Il 6502, diversamente dalla maggior parte degli altri microprocessori, non permette all'accumulatore di essere usato per operazioni d'incrementazione e decrementazione. Incrementare significa aggiungere 1, mentre decrementare significa sottrarre 1. Queste operazioni possono essere eseguite solo su byte contenuti nella memoria, così che il programmatore del 6502 deve incrementare o decrementare i byte prima o dopo il loro uso. Ciò può voler dire che un byte deve essere riposto nella memoria per essere incrementato o decrementato. È molto facile, tuttavia, fare uso di un ADD immediato per eseguire un incremento. Una sottrazione immediata non è così semplice, perché il flag carry nel registro P (Processor Status Register) deve essere azzerato prima di poter eseguire una sottrazione. Se non si azzera questo flag, invece di sottrarre 1 sottrarrete 2. L'accumulatore, tuttavia, può essere usato per importanti istruzioni di confronto.

Particolarmente utile è l'istruzione CMP (Compare), CMP è la mnemonica che deve fare uso di uno dei metodi standard di indirizzamento della memoria. Gli effetti di questa istruzione sono quelli di confrontare il byte copiato dalla memoria con quello già residente nell'accumulatore A. In questo senso, "COMPARE" significa che il byte copiato dalla memoria viene sottratto dal byte presente nell'accumulatore. La differenza tra questo tipo di sottrazione e una sottrazione vera e propria è che il risultato non viene memorizzato da nessuna parte, ma viene usato per settare i flag nel registro P e il byte nell'accumulatore rimane immutato. Ad esempio, supponiamo che l'accumulatore contenga il byte $4F, e che ci capiti di avere lo stesso tipo di byte memorizzato all'indirizzo $327F. Se usiamo l'istruzione:

CMP $327F

allora il flag zero nel registro CC viene settato (a 1), ma il byte nell'accumulatore rimane $4F, e il byte nella memoria $4F. La sottrazione avrebbe fatto sì che il contenuto dell'accumulatore risultasse uguale a zero.

Ma perché dovrebbe essere importante questa cosa?

Supponiamo che vogliate che il programma esegua una cosa se si preme il tasto "Y", e un'altra se si preme la lettera "N". Se sistemate le cose in modo tale che il programma in linguaggio macchina memorizzi il codice ASCII relativo al tasto premuto all'interno dell'accumulatore, lo potrete confrontare. Confrontandolo con $4E (il codice ASCII di "N"), si scopre se è stato premuto il tasto della lettera "N". Se così è avvenuto

viene settato il flag zero, se no, si può verificare ancora. Confrontandolo con $59, si può verificare se è stato premuto il tasto "Y". Verrebbe così settato di nuovo il flag zero. Se nessuno di questi confronti setta il flag zero, sapremo che le lettere "N" e "Y" non sono state premute, e potremo tornare indietro per nuovi raffronti. Se ciò vi sembra molto simile all'operazione che potete programmare usando il ciclo GET$ in BASIC, avete ragione, perché è proprio così.

Per finire, si hanno le operazioni di controllo e salto. Queste, come il nome suggerisce, permettono ai flag del registro P di essere controllati e di fare saltare il programma a un nuovo indirizzo se il flag è stato settato. Ma quale flag? Dipende dal tipo di istruzione di controllo e salto usata, perché ve n'è una per ogni flag principale, e per ogni stato del flag. Ad esempio, considerate i due test le cui mnemoniche sono BEQ e BNE. BEQ significa "salta se uguale a 0". Come essa ci suggerisce, causa un salto se il risultato della sottrazione o del confronto è 0. In altre parole, causa il salto se il flag zero è uguale a 1. Al contrario, BNE significa "salta se non uguale a 0" e farà sì che venga effettuato il salto se il flag zero non è settato. Esistono, perciò, due istruzioni di salto che controllano il flag zero, ma in maniera contraria. Lo stesso tipo di cose vale per molti altri flag.

BCC: branch on carry clear: salta a un nuovo indirizzo se il flag carry è azzerato.
BCS: branch on carry set: salta a un nuovo indirizzo se il flag carry è settato (a 1).
BEQ: branch if equal to zero: salta a un nuovo indirizzo se il flag zero è settato (a 1).
BNE: branch if not equal to zero: salta a un nuovo indirizzo se il flag zero non è settato (flag zero uguale a 0).
BMI: branch on result minus: salta a un nuovo indirizzo se il risultato dell'operazione precedente è stato un numero negativo (flag N azzerato).
PBL: branch on result positive: salta al nuovo indirizzo se il risultato dell'operazione precedente è stato un numero positivo, oppure zero (flag N azzerato).
BVC: branch on overflow clear: salta a un nuovo indirizzo se il flag overflow è 0.
BVS: branch if overflow set: salta a un nuovo indirizzo se il flag overflow è settato. Ciò accade se l'operazione di sottrazione o di addizione fa sì che il bit del segno (bit più significativo) cambi scorrettamente.

Nota: Tutte le istruzioni sopra elencate usano l'indirizzamento relativo del PC. Il byte dell'istruzione deve essere seguito da un solo byte, chiamato scostamento, che viene aggiunto all'indirizzo nel registro PC per ottenere il nuovo indirizzo.
JMP: jump to new address: salta al nuovo indirizzo fornito dai due byte che seguono il codice di JMP.

Figura 5.3 - L'elenco completo delle istruzioni di salto del 6502

Esiste anche un altro tipo di istruzione di salto, la mnemonica JMP, che non esegue nessuna verifica come l'istruzione GOTO, prima della quale non appare alcun IF.

L'elenco completo di tutte le istruzioni di salto a disposizione è mostrato nella figura 5.3. Molte di queste probabilmente non le userete mai, e le veramente importanti sono quelle che utilizzano il flag zero, il carry e quello negativo. Tutte, tranne JMP, fanno uso dell'indirizzamento relativo. Ciò significa che il codice per il salto deve essere seguito da un byte. In altre parole, questo numero viene trattato come numero segnato (sarà cioè negativo se supera il valore decimale di 127 — $7F—) e viene aggiunto all'indirizzo che è presente nel PC nel momento in cui viene effettuato il salto. Il risultato di questa somma è l'indirizzo a cui salterà il programma perciò, la nuova istruzione eseguita sarà quella che parte da questo nuovo indirizzo. Questo tipo di salto, che utilizza un numero dallo scostamento di un solo byte, permette uno spostamento fino a 127 (dec.) posizioni in avanti o 128 indietro. Questo perché un singolo byte segnato non può superare tali valori.

Operazioni sul Commodore 64

È giunto il momento di far pratica con alcuni programmi in linguaggio macchina. Non si tratta solo di battere le linee del programma come se fossero scritte in BASIC. A meno che non abbiate un programma assemblatore su cartuccia, il C64 ci invierà semplicemente il messaggio di errore "SN ERROR" ogni volta tenterete di eseguirlo. Dato che è nostra intenzione iniziare dalle cose semplici, lasceremo da parte gli assemblatori e assembleremo "a mano". Ciò significa che dovremo cercare i codici del linguaggio macchina che corrispondono alle istruzioni del linguaggio assemblatore su una tavola. Quindi, dovremo tradurre i codici esadecimali e i dati in numeri decimali. Li immetteremo poi all'interno della memoria del computer usando l'istruzione POKE, ponendo il primo byte nel PC del 6502 e osservando cosa accade. Sembra semplice, ma ci sono molte cose a cui pensare e alcune precauzioni da prendere.

Come abbiamo visto, il C64 utilizza gran parte della propria RAM per i propri compiti. Se si immette in una qualsiasi parte della memoria un certo numero di byte, può succedere di sostituire i byte di cui necessita il C64, oppure saranno i nostri byte di programma ad essere sostituiti dalle operazioni del C64. Abbiamo perciò bisogno di una parte di memoria protetta soltanto per nostro uso.

Ciò può essere fatto sfruttando la possibilità che ha il C64 di spostare i suoi programmi in BASIC allo stesso modo in cui può spostare la VLT. Lo spostamento più facile da eseguire è la fine della memoria. L'ultimo byte della RAM che può essere occupato da un vostro programma è il 40960 (dec.). Gli indirizzi alla destra della fine della memoria vengono normalmente usati per memorizzare delle stringhe non presenti prima nella memoria: lo abbiamo già visto nel Capitolo 2. Ora, il C64 non è programmato per fermarsi automaticamente a questo numero. È stato memorizzato "il numero di fine RAM", in due byte, nelle locazioni della memoria 55 (byte basso) e 56 (byte alto). Questi sono indirizzi di pagina zero, e durante lo svolgimento di un programma BASIC il computer controllerà continuamente di non star usando locazioni superiori al numero memorizzato in essi.

Supponiamo di cambiare questi numeri. Non è necessario alterare entrambi i byte perché, se riduciamo di 1 il byte alto, ne avremo a nostra disposizione ben 256. Ciò accade perché un numero memorizzato all'interno di questi due byte è 256 volte il byte alto più il byte basso. Quando viene acceso il C64, il numero memorizzato nell'indirizzo 56 è 160. Battendo POKE 56,159 riserveremmo 256 byte, partendo dall'indirizzo 40705 e arrivando fino al 40960. Ricordate che sono inclusi il primo e l'ultimo byte. Una volta fatto ciò, possiamo essere certi che i programmi BASIC non utilizzeranno gli indirizzi superiori al 40704, e perciò non potranno interferire con i nostri programmi in linguaggio macchina.

L'altro problema è il posizionamento dell'indirizzo di partenza del programma all'interno del 6502. Per fortuna, i progettisti del 6502 sono stati cortesi e hanno predisposto un comando BASIC SYS che svolge questa operazione. SYS deve essere seguito da un numero, che verrà messo nel PC: verrà quindi usato come indirizzo del primo byte del vostro programma. Per inciso, ho considerato questo indirizzo come "indirizzo di partenza" anche se avrei potuto usare i primi byte per memorizzarvi dei dati, facendo iniziare il programma, ad esempio, al decimo byte. Ciò non crea alcun problema: dovete semplicemente usare il byte di partenza come numero per il comando SYS. Infine, almeno per il momento, dovete assicurarvi che il programma si fermi in modo ordinato. Nulla, fino a questo punto, ha indicato al 6502 dove finisce il vostro programma. Come risultato, il 6502 continua a leggere i byte anche dopo l'esecuzione, fino a quando non incontra un byte che causa un suo blocco. Questo potrebbe essere, ad esempio, un byte che crea un ciclo senza fine. Alcuni programmatori dubitano del fatto che possa esistere un byte che non causi un ciclo senza fine in casi come questi! Per tornare correttamente al sistema operativo del C64, bisogna finire ogni programma in linguaggio macchina usando una istruzione "ritorno da subroutine", la cui mne-

monica è RTS e il cui codice è $60.

Un altro problema di cui al momento non dobbiamo preoccuparci è che, quando si esegue un programma in linguaggio macchina con un programma BASIC sul C64, si usa per entrambi i compiti lo stesso microprocessore 6502 che, non potendo fare due cose contemporaneamente, ne esegue prima uno, poi l'altro. Usando i registri del 6502 nel programma in linguaggio macchina, come siete costretti a fare, dovete accertarvi di non stare distruggendo delle informazioni di cui ha bisogno il programma in BASIC. Ad esempio, se nell'istante in cui avete eseguito il programma in linguaggio macchina i registri del 6502 contengono l'indirizzo di una parola riservata nella ROM, il microprocessore avrà bisogno di questi alla fine del vostro programma in linguaggio macchina.

Quando un programma in linguaggio macchina entra in funzione tramite il comando SYS, ci si occupa di ciò automaticamente. I contenuti dei registri del 6502 vengono messi in una parte della RAM chiamata "stack" (pila). Questo, per inciso, è un altro buon motivo per fare attenzione al punto in cui mettere il vostro programma in linguaggio macchina. Se cancellate lo stack, il C64 non sarà più lo stesso. Lo stack si trova dalla locazione 256 fino alla 511. Incontrando l'istruzione RTS alla fine del programma in linguaggio macchina, i byte memorizzati all'interno dello stack vengono posti di nuovo nei loro registri e si ripristinano le normali funzioni. Se eseguite un programma in linguaggio macchina usando un altro metodo, senza l'uso di SYS, dovrete fare da soli queste operazioni come parte del programma in linguaggio macchina. Ciò implica l'uso dei comandi PUSH e PULL, che vedremo meglio in seguito.

Un programma pratico

Avendo finito i preliminari, possiamo iniziare con qualche programma molto semplice, che ha lo scopo di spiegare il modo in cui i programmi sono memorizzati all'interno della memoria del C64. Potrete esercitarvi nell'uso del linguaggio assembler e in quello macchina, e sul modo in cui si esegue un programma in linguaggio macchina.

Inizieremo con l'esempio più semplice possibile: un programma per mettere nella memoria un byte. In linguaggio macchina viene scritto:

```
ORG 40705; inizia a mettere i byte qui.
LDA #$55; metti 55 esadec., nell'accumulatore.
STA $9FC4; memorizzali all'indirizzo 9FC4.
RTS; torna al BASIC.
```

La prima linea contiene una mnemonica, ORG, che vedete per la prima volta. Essa non fa parte delle istruzioni del 6502, ma è una istruzione per l'assembler, che in questo caso siete voi! ORG è l'abbreviazione di origine, e sta a ricordarvi che questo è il primo indirizzo ad essere usato per il vostro programma. Abbiamo scelto di usare un indirizzo che lasci spazio a programmi più lunghi di quelli che scriveremo nel corso di questo libro, ma avremmo potuto scegliere un numero più alto. Tuttavia, andrà bene come qualsiasi altro e permette di avere molto spazio per programmi più lunghi.

Quando si programma usando l'assembler, si può battere questa linea e l'assembler posizionerà automaticamente i byte del programma all'interno della memoria partendo da questo indirizzo. In effetti, scrivendo in linguaggio macchina, esso ci ricorda semplicemente quale indirizzo usare. Notate i commenti dopo i punti e virgola. I punti e virgola rivestono nel linguaggio assembler la stessa funzione svolta dall'istruzione REM in BASIC. Tutto quello che segue un punto e virgola è solo un commento che l'assembler ignora, ma che può essere utile per il programmatore.

Osserviamo ora il programma. La prima vera istruzione è di caricare il numero $55 nell'accumulatore "A". Questo utilizza l'indirizzamento immediato, così il numero $55 dovrà essere posto immediatamente dopo l'istruzione. Il simbolo "#", castelletto, viene usato in linguaggio macchina per indicare che si deve usare un caricamento immediato (diretto). Il comando successivo ordina di memorizzare il byte nell'accumulatore (ora $55) all'indirizzo $9FC4. In binario, equivale a 40900 ed è un indirizzo molto al di sopra di quelli usati dal programma. Ovviamente, non utilizzeremo un indirizzo che potrebbe venir usato dal programma. Questa istruzione usa l'indirizzamento assoluto. Infine, il programma termina con l'istruzione RTS, essenziale per far procedere il C64 normalmente dopo la fine del nostro programma.

Il passo successivo nella programmazione è la trascrizione dei codici in esadecimale. Si deve consultare ciascun codice, avendo cura di scegliere quello esatto per il metodo di indirizzamento. Il codice per LDA immediato è $A9, perciò questo è il primo byte del programma ad essere memorizzato all'indirizzo 40705 (dec.). Possiamo iniziare una tavola di indirizzi e dati numerici usando

40705 $A9

quindi procedere. Il byte che vogliamo caricare è $55 e deve essere messo nel successivo indirizzo della memoria, poiché è così che l'indirizzamento

immediato funziona. La tavola, ora, appare come segue:

40705 $A9
40706 $55

Il successivo byte di cui abbiamo bisogno è il byte dell'istruzione STA, con indirizzamento assoluto. Questo byte è $80 e deve essere seguito dai due byte dell'indirizzo in cui vogliamo memorizzarlo. L'indirizzo 40900 deve essere tradotto in codice esadecimale $9FC4, in modo che potremo utilizzare i byte $C4 e $9F che seguono l'istruzione STA. Ricordate che questi byte devono essere ordinati secondo la sequenza LO-HI (basso-alto). L'ultimo codice deve essere quello RTS ($60), che fa apparire la tavola come in fig. 5.4. Esso usa gli indirizzi dal 40705 al 40710, 6 in tutto, e mette un byte all'interno del 40900, usando numeri decimali. Ora dobbiamo metterlo nella memoria e farlo funzionare!

40705	A9
40706	55
40707	8D
40708	C4
40709	9F
40710	60

Figura 5.4 - Il programma codificato usando indirizzi in decimale e i byte dei dati in esadecimale

È necessario un programma in BASIC che liberi la memoria e che immetta i byte uno alla volta con l'istruzione POKE. Prima di poterlo scrivere dobbiamo convertire ogni byte esadecimale nella forma decimale, perché l'istruzione POKE del C64 utilizza i soli numeri in notazione decimale. Potete convertire tali numeri usando una calcolatrice o per mezzo del programma illustrato nel capitolo 3. Il programma BASIC con POKE è illustrato nella figura 5.5. Tramite POKE56, 159, ci assicuria-

```
10 POKE 56,159:A=40704
20 FORN=1TO6:READ D%
30 POKEA+N,D%:NEXT
40 SYS 40705
100 DATA169,85,141,196,159,96
```

Figura 5.5 - Il programma Basic con cui si utilizza POKE. Da notare come il numero intero D% è stato usato per i dati e come il SYS40705 faccia eseguire il programma

mo che gli indirizzi della memoria, superiori a 40704, non vengano utilizzati dal C64. Dichiariamo la variabile A come 40704, in modo tale da poterla utilizzare nei comandi POKE. Le linee 20, 30 e 40 immettono i numeri che seguono l'istruzione data negli indirizzi, a partire dal 40705. Perché 40705?

Abbiamo usato POKEA+N, e con A=40704 e N=1, il primo indirizzo può solo essere 40705. Tutti i codici devono essere nella notazione decimale per poter essere utilizzati dall'istruzione POKE all'interno del programma BASIC; i problemi nascono solo quando si ha un numero dell'indirizzo in decimale e lo si deve convertire in due byte del numero decimale. La procedura è mostrata nell'appendice B, prima della conversione decimale. È meglio, tuttavia, lavorare in esadecimale il più possibile, e convertire in decimale solo quando si è costretti. Dato che dovete fare uso dell'esadecimale con l'assembler MIKRO o qualsiasi altro assembler, è meglio impararlo bene.

L'ultima linea del programma, la linea 40, contiene SYS40705. Questa è l'istruzione BASIC per eseguire il programma in linguaggio macchina, con l'indirizzo di potenza specificato. La linea 100 contiene i 6 byte dei dati che abbiamo calcolato. Eseguendo (RUN) il programma apparentemente non avrete nessun effetto. Questo perché non potete vedere quanto è contenuto all'indirizzo 40900. Usando

```
?PEEK(40900)
```

dovreste ottenere il valore 85, che è la versione in decimale di $55, il numero messo là dal programma.

Ora proviamo questo: battete POKE 40900,255, premete RETURN, e cancellate la linea 50 dal vostro programma. Questa è la linea relativa all'istruzione SYS. Eseguite (RUN) di nuovo il programma, usando ?PEEK(40900) per vedere che cosa si trova là. Dovrebbe essere 255.

Ora battete SYS40705 e premete RETURN. Usando ?PEEK(40900) dovreste di nuovo ottenere 85. Ciò accade perché l'immissione dei byte del programma nella memoria non farà eseguire il programma in quanto solo SYS lo può fare. Si possono perciò immettere i valori all'interno della memoria in un programma BASIC per poi farne uso più tardi tramite SYS quando lo si desidera.

Questo programma non intende essere un'opera d'arte: non esegue altro che la semplice istruzione POKE40900 (85 in programma BASIC), ma è un inizio. La cosa più importante a questo punto è di abituarsi al modo in cui il linguaggio macchina opera, come metterlo nella memoria ed eseguirlo. Un altro punto, per inciso, è che il linguaggio macchina è al sicuro nella memoria. Se digitate NEW (RETURN), il programma BASIC verrà cancellato, ma le istruzioni in linguaggio macchina riman-

gono intatte. Se digitate POKE40900, ora 255, e verificate con ?PEEK (40900), troverete che questo indirizzo può essere ancora cambiato tramite l'istruzione SYS40705. Questi byte rimarranno là fino a quando non li cambierete o li azzererete spegnendo il computer. Se lo desiderate potrete salvare il programma su nastro, ma questa è una tecnica di cui parleremo in seguito. Un passo alla volta, per favore! Un'altra cosa che tratteremo più tardi è il metodo alternativo di richiamare un programma usando l'istruzione USR.

Ora proviamo qualcosa di più ambizioso in termini d'uso del linguaggio macchina, sebbene l'esempio sia abbastanza semplice. La figura 5.6 mostra la versione del programma in linguaggio assembler. Quello che faremo sarà caricare un byte nell'accumulatore, spostarlo di una posizione a sinistra per poi metterlo nella memoria all'indirizzo di un passo maggiore di quello da cui lo si è preso.

```
LDX #$0          A2 00
LDA $9FC$X       BD C4 9F
ASL A            0A
INX              E8
STA $9FC4,X      9D C4 9F
RTS              60
```

Figura 5.6 - Il programma in linguaggio assembler per "moltiplicare per due". L'elenco mostra il linguaggio assembler sulla sinistra e i codici esadecimali sulla destra

Assomiglia a un caso di apertura e chiusura per l'indirizzamento indicizzato, perciò dovremo partire mettendo uno 0 nel registro X. Questo è il passo LDX #$00. Come prima, il simbolo "#" significa indirizzamento immediato. La linea successiva, LDA $9FC4,X significa che l'accumulatore sta per essere caricato del contenuto dell'indirizzo $9FC4, maggiorato del numero contenuto nel registro X. Il terzo passo, ASL A, effettua uno spostamento aritmetico a sinistra del byte contenuto nell'accumulatore in modo tale da spostare a sinistra i bit del byte in esso contenuto. Quindi, viene incrementato il numero all'interno del registro X tramite INX. Ciò trasforma lo 0 in 1. Ora possiamo immagazzinare il byte nell'accumulatore all'indirizzo $9FC5 usando STA 9FC4,X. Questa volta, poiché è stato aggiunto 1 al numero contenuto nel registro X, il byte viene memorizzato all'indirizzo $9FC5. Si finisce, come sempre, con l'istruzione RTS.

Ora possiamo scrivere tutto questo in linguaggio macchina. Tale procedimento non è più difficile di quello precedente, nonostante l'uso del-

l'indicizzazione. L'istruzione LDX ha bisogno del caricamento immediato del codice $A2 che deve essere seguito dal byte di dati, $00. L'istruzione LDA con l'indirizzamento indicizzato è codificato come $BD e deve essere seguito dai due byte dell'indirizzo $9FC4, nel loro solito ordine byte basso-alto. Il codice per l'istruzione ASL è 0A, quindi eseguiamo l'incremento di X usando INX, il cui codice è $E8. Quindi memorizziamo il byte che è nell'accumulatore di nuovo nella memoria usando STA $9FC4,X. Il codice X indicizzato dell'istruzione STA è $9D, seguito dai byte dell'indirizzo. Per finire, $60 è il comando RTS. Ora dobbiamo codificarlo in BASIC.

Se scegliamo un numero piccolo da mettere all'indirizzo $9FC4, l'effetto dello spostamento a sinistra sarà di raddoppiare il numero stesso, perciò possiamo usarlo per ottenere un po' di aritmetica 'magica'.

```
10 POKE 56,159:A=40704
20 FOR N=1 TO 11:READ D%
30 POKEA+N,D%:NEXT
40 POKE 40900,7:SYS 40705
50 PRINT ;PEEK(40900);" AL POSTO DI ";PEEK(40901)
100 DATA162,0,189,196,159,10,232,157,196,159,96
```

Figura 5.7 - Il programma BASIC che sistema i byte al loro posto tramite l'istruzione POKE per poi fare uso del programma in linguaggio macchina

Il programma in BASIC è mostrato nella figura 5.7. Iniziamo, come al solito, azzerando lo spazio di memoria. Non dovete preoccuparvi se prima avevate un programma in questa parte della memoria. Il nuovo lo sostituirà completamente e, a patto che termini con l'istruzione RTS, i vecchi programmi non possono interferire con il nuovo. I valori vengono immessi al loro posto nel solito modo dalla linea 20 alla 30. Nella linea 40 mettiamo un numero, 7 decimale, all'indirizzo $9FC4 (40900 dec.). Questo è l'indirizzo che verrà usato dal programma e il byte, che è 7, verrà messo in questo indirizzo nella forma binaria 00000111. Nella seconda parte della linea 40, SYS 40705 eseguirà il programma in linguaggio macchina, che dovrebbe spostare a sinistra questo byte rendendolo uguale a 00001110. In decimale, questo è 14, il doppio di 7. La linea 50 stampa il risultato, mentre la linea 100 contiene i byte dei dati.

È abbastanza semplice, ma se non conosceste nulla del linguaggio macchina vi chiedereste come mai il numero viene moltiplicato per due. Di nuovo, il programma non fa niente che non possa essere fatto più facilmente e rapidamente usando il linguaggio BASIC. Ciò che importa, dal nostro punto di vista, è che ora avete usato l'indirizzamento indicizzato

e una istruzione di spostamento mentre, nel contempo, avete fatto l'esperienza di mettere un programma in linguaggio macchina all'interno del vostro C64 con il metodo più difficile. Se, per caso, avete compiuto qualche errore, in particolar modo nei DATA, è probabile che il computer si rifiuti di eseguire qualsiasi cosa. Dopo aver battuto un programma in BASIC, ricordatevi sempre di salvarlo prima di eseguirlo. In questo modo, se un byte scorretto paralizza metà della RAM, non dovrete fare altro che spegnere il computer, per poi riaccenderlo, e ricaricare il vostro programma. Se non lo avete salvato prima, dovrete riscriverlo di nuovo per intero. Questo è un compito faticoso, e la vita è già abbastanza faticosa di per sé.

Per finire, un altro punto riguarda il numero immesso nella locazione 40900. Se è piccolo, allora il programma funziona; se supera 127, otterrete uno strano risultato poiché viene trattato come se fosse negativo; in ogni caso non potete immettere un numero superiore a 255 in una qualsiasi singola posizione di memoria. Le routine per moltiplicare i numeri nel C64 sono comunque molto più sofisticate di quanto non sembri da questo esempio!

Capitolo 6

La programmazione

I programmi presi in considerazione nel capitolo 5 non fanno molto, però sono utili per capire come sono scritti i programmi in linguaggio macchina. È essenziale esercitarsi col linguaggio assemblatore e con la sua conversione in linguaggio macchina e si possono capire meglio gli errori se i programmi sono così semplici.

Non è affatto facile trovare un errore nei programmi in linguaggio macchina più lunghi, soprattutto quando state imparando tale linguaggio.

Abbastanza stranamente, nascono molte difficoltà per i principianti, nonostante il linguaggio macchina sia semplice. Poiché è semplice, sono necessari vari passi del programma per poter raggiungere un qualsiasi scopo utile. La parte più difficile della programmazione consiste nel dividere quello che volete fare in tanti passi eseguibili dalle istruzioni in linguaggio macchina. Per questa parte della programmazione, i diagrammi di flusso sono di gran lunga il metodo più utile per potersi orientare. Non ho mai pensato che i diagrammi di flusso siano l'ideale per la programmazione in BASIC, ma certamente lo sono nella programmazione in linguaggio macchina.

I diagrammi di flusso

I diagrammi di flusso sono per il software. Essi mostrano ciò che deve essere fatto o (tentato) senza per questo entrare nei dettagli. Un diagramma di flusso è formato da una serie di simboli, ognuno dei quali rappresenta un tipo di operazioni. La figura 6.1 ne mostra alcuni tra i più importanti per i nostri scopi (presi dal set di simboli standard per i diagrammi di flusso). Questi sono i simboli terminatori (inizio o fine), di procedura, di input/output e di decisione. All'interno dei simboli stessi, si possono scrivere brevi note sul tipo di operazione desiderata, ma senza entrare nei dettagli. Un esempio vale sempre di più di tante spiegazioni.

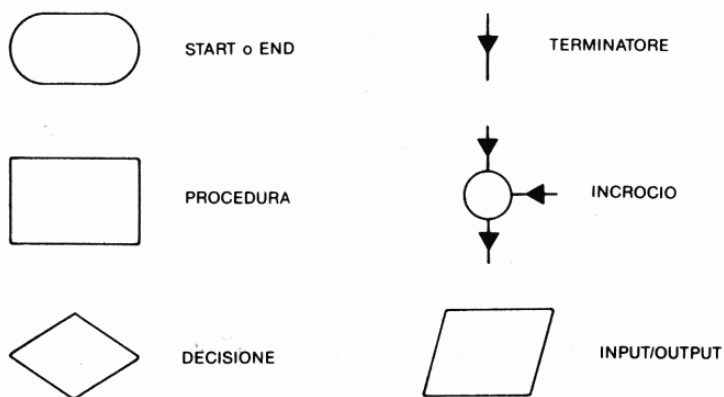



Figura 6.1 - Principali diagrammi di flusso

Supponiamo di volere un programma in linguaggio macchina che prenda il codice ASCII dal tasto premuto e stampi il carattere corrispondente a quel tasto. Un diagramma che descrive questa operazione è illustrato nella figura 6.2. Il primo simbolo è quello di inizio programma (START) poiché ogni programma o parte di esso deve iniziare da qualche parte.

Figura 6.2 - Un diagramma di flusso per il programma "stampa un carattere"

La linea con la freccia mostra che si passa al primo blocco operativo, con il commento "prendere il carattere e metterlo in A"; Questo commento ci dice quello che intendiamo fare, e cioè prendere il codice numerico in un carattere e metterlo nell'accumulatore. Dopo aver messo il carattere nell'accumulatore, la freccia indica l'operazione successiva, cioè memorizzare il byte nella memoria dello schermo. Questo è il modo in cui si esegue la parte della stampa dell'operazione che abbiamo già visto. Il terminatore END ci ricorda che qui finisce il programma che, perciò, non è un ciclo senza fine.

Questo è un diagramma di flusso estremamente semplice, ma è sufficiente per illustrare quanto voglio dire. Notate che le descrizioni sono piuttosto generiche. All'interno dei simboli di un programma di flusso non si mettono le istruzioni in linguaggio assembler. A rigor di termini, non bisognerebbe riferirsi all'accumulatore A nel riquadro della prima operazione; pur tuttavia aiuta a ricordare il luogo in cui memorizzare il codice.

Un diagramma di flusso dovrebbe essere scritto in modo tale che mostri, a chiunque lo guardi, quello che sta succedendo; non dovrebbe essere solo una cosa comprensibile all'analista e che invece confonde qualsiasi altra persona. Un buon diagramma di flusso, infatti, può essere usato da qualsiasi programmatore per scrivere un programma in ogni tipo di linguaggio macchina, o in ogni altro linguaggio come il Basic, Forth, Pascal, eccettera. Molti diagrammi di flusso, purtroppo, sono costruiti dopo che il programma è stato scritto (di solito dopo molti tentativi ed errori), con speranza di rendere più comprensibile l'operazione. Ma non è così.

Quando avete il diagramma, potete controllare che esegua ciò che desiderate riguardandolo con estrema attenzione. Nell'esempio, le operazioni del "prendere il carattere" e "memorizzarlo nella memoria dello schermo" verranno svolte in linguaggio macchina, perciò ci concentreremo su di esse. Ottenere il codice ASCII di un carattere, a prima vista può sembrare complicato. Molti computer, tuttavia, mettono il codice ASCII dell'ultimo carattere usato in un indirizzo della memoria. Qui torna utile una buona conoscenza delle modalità di uso della memoria da parte del C64! L'appendice E mostra alcuni di questi importanti indirizzi. Uno di essi, di interesse particolare, è il 512 decimale ($200): è l'indirizzo d'inizio del "buffer di tastiera". Il buffer (memoria di transito) è una sezione della memoria RAM che il computer utilizza per memorizzare i dati temporaneamente. Come il nome suggerisce, il buffer di tastiera viene utilizzato per memorizzare i numeri dei codici dei tasti che avete premuto. Questi codici rimangono nel buffer fino a quando non viene premuto il tasto RETURN. Quando si preme questo tasto, il contenuto del buffer

viene spostato in un'altra parte della memoria. L'importanza del buffer è che può essere usato per sapere quale tasto è stato premuto per ultimo. Se si carica l'accumulatore del contenuto del buffer, si possono ricevere delle informazioni molto utili. Il primo passo è quindi caricare l'accumulatore, usando l'indirizzamento esteso, così da poter far uso dell'indirizzamento 512.

Il secondo passo, la registrazione del byte nella memoria dello schermo, è semplice. La memoria dello schermo utilizza gli indirizzi a partire dalla locazione 1024 fino alla locazione 2023 (decimale). Se volete, ad esempio, visualizzare un carattere al centro dello schermo, non dovete fare altro che registrare nella locazione numero 1524 il codice del simbolo grafico voluto. Poiché gli indirizzi della memoria dello schermo sono 1000 (2023—1024), non dovremo fare altro che aggiungere 500 a 1024 (=1524).

Ora possiamo stendere la parte del codice in linguaggio assemblatore. Si può usare la procedura precedente e iniziare il codice all'indirizzo 40705 (decimale). Perciò avremo:

```
ORG 40705
LDA $0200
STA $05F4
RTS
```

Ora possiamo calcolare a mano i codici in esadecimale, i quali sono:

```
AD 0002
8D F405
96
```

Se convertiamo questi valori in decimale, avremo i codici utilizzabili in un programma che usi POKE all'interno della memoria dello schermo. Possiamo usare questi codici in un programma BASIC tramite POKE e quindi richiameremo il programma in linguaggio macchina. Esso apparirà come in fig. 6.3. Non dobbiamo fare altro che digitarlo ed eseguirlo. Bene, funziona, ma non come ci saremmo aspettati e il programma mostra ciò che si sarebbe dovuto aggiungere. Per iniziare si deve trovare il modo di mettere un codice nel primo indirizzo del buffer di 512 ($0200). Per compiere questa operazione, il computer utilizza il ciclo GET A$ alla linea 40. Il secondo punto è che sullo schermo non appare nulla fino a quando non si dà un colore al carattere che si vuole visualizzare, e ciò accade alla linea 60. Se battete questo programma e lo fate eseguire, funzionerà. Per farlo funzionare di nuovo non dovrete far altro che premere il tasto STOP/RESTORE per ripristinare i registri del colore.

```
10 POKE 56,159:A=40704
20 FORN=1TO7:READ D%
30 POKEA+N,D%:NEXT
40 GET A$:IF A$=""THEN 40
50 SYS 40705
60 POKE 53281,243
100 DATA173,0,2,141,244,5,96
```

Figura 6.3 - Il programma BASIC per la stampa di un carattere

Se non lo farete, il programma agirà ugualmente, ma non potrete vederne gli effetti sullo schermo.

Allo stato attuale delle nostre conoscenze, programmiamo il ciclo in BASIC, poi richiamiamo il programma in linguaggio macchina non appena il BASIC ha rivelato che è stato premuto un tasto. Invece di perdere altro tempo con un metodo "misto" come questo, guardiamo come si fa a stendere l'intero programma in linguaggio macchina, anche rischiando di commettere errori strada facendo.

Cicli in attesa

Dato che questo è un programma semplice, è opportuno introdurre il concetto di ciclo. Se avete programmato in BASIC in modo abbastanza evoluto, saprete cosa implica un ciclo. Un ciclo non è altro che la ripetizione di una determinata parte di un programma fino a quando non accade una determinata cosa. Nel BASIC si può creare un ciclo utilizzando una linea comune:

200 IF A=0 THEN GOTO 100

Questo ciclo contiene un test (A=0?). Se il test è verificato (A=0), allora il controllo salta indietro alla linea 100 per ripetere le istruzioni da questa linea alla linea 200. Questo tipo di ciclo in BASIC assomiglia molto al modo in cui se ne crea uno in linguaggio macchina. Invece di fare uso di numeri di linea, tuttavia, si fa uso di numeri di indirizzi. Invece di verificare la variabile chiamata "A" si verifica il contenuto di un registro, che in questo caso è l'accumulatore.

Iniziamo nel modo appropriato con un diagramma di flusso. La fig. 6.4 mostra come dovrebbe apparire. Il primo passo è identico: mette il codice del carattere nell'accumulatore A. Il passo successivo è una operazione di "decisione". La decisione è: "è 0?" Tutti i passi decisionali nei diagrammi di flusso devono essere marcati in modo tale da ottenere

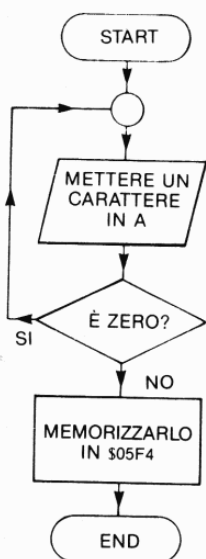



Figura 6.4 - Un altro diagramma di flusso che contiene una iterazione che rifiuta il carattere zero

due sole possibili risposte: sì o no. Ciò si ottiene graficamente tramite due percorsi (frecce) che partono dal punto della decisione. Uno di questi percorsi viene marcato 'SI'. Esso ritorna indietro al primo passo del programma, il passo che carica nell'accumulatore il contenuto della memoria. Questo perché? Se il contenuto dell'accumulatore è 0, significa che non è stato premuto il tasto e che bisogna riprovare. L'altro percorso, marcato "NO", indica l'istruzione successiva facendo registrare nella memoria dello schermo il byte contenuto nell'accumulatore.

L'operazione, quindi, sarà di caricare l'accumulatore del contenuto della locazione 512, e di verificare se il byte nell'accumulatore è uguale a zero e, in caso contrario, di ripetere il caricamento. Se il byte è diverso da 0, allora significa che è stato premuto un tasto. Questo valore verrà registrato nella memoria dello schermo. Passiamo ora alla stesura di questi passi in linguaggio macchina: introdurremo nuove istruzioni e nuovi problemi.

La fig. 6.5 mostra un programma in linguaggio assembler che dovrebbe ottenere i risultati previsti nel diagramma di flusso. In questo diagramma è presente un passo in più e una variazione in uno già esistente. Il nuovo passo è il 'BEQ LOOP' e la variazione è nel primo passo, che

```
CICLO      LDA $0200
           CICLO BEQ
           STA $05F4
           RTS
```

Fig. 6.5 - Il programma in linguaggio assemblatore corrispondente al diagramma di flusso

ora ha scritta davanti la parola "LOOP" (CICLO). La parola "CICLO" è una "etichetta". Qui viene usata al posto di un indirizzo, e rappresenta la locazione della memoria da cui inizia l'istruzione. In questo caso, CICLO LDA 512 rappresenta l'indirizzo della memoria in cui è immagazzinata l'istruzione LDA. Usando le parole in questo modo, non dobbiamo preoccuparci dei numeri degli indirizzi fino a quando non scriviamo effettivamente in linguaggio macchina. Se si usa un assemblatore, di solito non ci si deve preoccupare degli indirizzi; l'assembler li colloca al posto delle parole mancanti. La stessa "etichetta" viene usata nel passo successivo. BEQ significa "salta se il registro è uguale a 0", così BEQ LOOP produce l'effetto di spostare il programma all'indirizzo dell'istruzione LDA se l'accumulatore è azzerato. È come l'uso di certe versioni del BASIC che permettono l'utilizzo di variabili al posto di numeri di linea.

Nel linguaggio assemblatore, questa cosa sembra chiara e semplice. Se stessimo usando un linguaggio assembler, lo sarebbe, ma se l'assemblaggio viene compiuto "a mano", non lo è. La ragione è che l'istruzione BEQ deve essere seguita da un'istruzione di un singolo byte che fornisca l'indirizzo dell'istruzione LDA. Questo è l'indirizzamento relativo del Program Counter, così che dobbiamo usare un byte segnato che può essere aggiunto all'indirizzo all'interno del PC per fornire l'indirizzo del passo dell'istruzione LDA. La formula è illustrata nella fig. 6.6. Bisogna trovare l'indirizzo al quale si vuole che il programma salti, e l'indirizzo del comando per il salto. Sottraendo questi numeri, quindi sottraendo 2 dal risultato, otterrete il valore dell'indirizzo di scostamento che

Destinazione: indirizzo al quale si salta (ha l'etichetta prima dell'istruzione assembler).
Sorgente: indirizzo da cui si salta (indirizzo del codice di salto — in linguaggio assemblatore — l'etichetta segue l'istruzione).
Scostamento: destinazione — sorgente — 2, in notazione esadecimale.

Figura 6.6 - La formula per calcolare il valore dell'indirizzo di scostamento

40705	173
40706	0
40707	2
40708	240
40709	indirizzo di scostamento

L'indirizzo sorgente è 40708, in cui si trova BEQ. L'indirizzo destinazione è 40705, l'istruzione LDA. La procedura è:
Indirizzo destinazione — indirizzo sorgente = 40705—40708 = —3; quindi si sottrae 2, così che —3—2=—5.
In decimale, l'equivalente byte per —5 è 256—5=251. Questo numero perciò deve essere il valore all'interno della locazione 40709.

Figura 6.7 - Un esempio di come viene calcolato il valore di scostamento

dovete fare seguire all'istruzione di salto. Dato che questo numero è negativo, dobbiamo convertirlo nella forma di byte segnato, usando le modalità viste in precedenza.

Se tutto questo può suonare complicato, osservatelo nella pratica, in figura 6.7. Supponendo di andare a mettere il primo byte del programma all'indirizzo 40705, l'indirizzo dell'istruzione BEQ è nella locazione 40708. L'indirizzo 50708 è l'indirizzo sorgente, dal quale partiamo, mentre 40705 è l'indirizzo destinazione, verso il quale andiamo. Se si sottrae il numero della sorgente da quello della destinazione si ottiene —3. Sottraendo al risultato 2, otterremo —5 che, in esadecimale, equivale a FB che è lo scostamento che deve seguire l'istruzione BEQ.

```
10 POKE 56,159:A=40704
20 FORN=1TO9:READ D%
30 POKEA+N,D%:NEXT
40 GET A$:IF A$="" THEN 40
50 SYS 40705
60 POKE 53281,243
100 DATA173,0,2,240,251,141,244,5,96
```

Figura 6.8 - Il programma BASIC per il linguaggio assembler della figura 6.5

Provando il programma illustrato nella fig. 6.8, noterete che funziona e che ci fornisce qualche indizio riguardo il Commodore 64. Molti computer non eseguirebbero un programma del genere e se ne avete capito il motivo, potete considerarvi degli assi. Ciò che accade è che il computer rimane all'interno di un ciclo fino a quando non viene immesso un codice numerico all'indirizzo 512, e un codice numerico viene immesso

quando viene premuto un tasto. Ora, su molti computer, se il microprocessore impiega il suo tempo all'interno del ciclo del vostro programma, non può controllare la tastiera alla ricerca di un eventuale tasto premuto! Nel Commodore 64 esiste un solo microprocessore che deve accollarsi entrambi i compiti. In effetti, esso non è solo occupato ad eseguire il vostro programma: ogni tanto interrompe ciò che stava facendo per controllare la tastiera.

Se si preme un tasto, allora il codice numerico di quel tasto viene messo nel buffer. Questo tipo di operazione viene chiamata, abbastanza propriamente, INTERRUPT (interruzione) ed è un'operazione del computer molto utile. L'utilizzazione di questa funzione rende più difficile gettare nello scompiglio il Commodore 64 in cicli eseguiti scorrettamente. Inoltre il microprocessore non deve mantenere in funzione ciò che appare sullo schermo — se così fosse eseguendo questo programma vedreste scomparire le immagini dallo schermo.

Esiste un altro modo? Quello che dobbiamo fare è di scrivere una parte di programma che si occupi di leggere la tastiera, per poi metterlo all'interno del nostro ciclo. È possibile, ma richiede molto tempo e una certa conoscenza del Commodore 64. Inoltre, risulta essere inutile perché nella ROM esiste già una routine che svolge questa funzione per noi. Nell'appendice E è riportata la lista degli indirizzi di questa routine e di altre routine utili. La routine che parte dall'indirizzo $E112 controlla la tastiera alla ricerca di un eventuale tasto premuto. Se ciò non accade, il valore del byte all'interno dell'accumulatore è 0. Se un tasto è premuto, allora nell'accumulatore viene registrato il valore del codice che rappresenta quel tasto. Usando questa routine, non si deve ricorrere all'uso della locazione di memoria come la 512.

Il passo è successivo è il vedere come utilizzare la routine che parte dalla locazione $E112. L'istruzione di cui abbiamo bisogno è quella per il salto a questa routine (JSR = jump to subroutine = salta alla subroutine). Ogni subroutine nella ROM termina coll'istruzione RTS, che sposta il controllo del computer indietro, al programma che aveva richiamato questa subroutine. Perciò, se usiamo JSR seguito dall'indirizzo $E112, allora verrà eseguita questa subroutine per poi passare di nuovo al programma principale. La figura 6.9 mostra un diagramma di flusso che rappresenta quello che cercheremo di fare ora.

Richiameremo la subroutine per registrare nell'accumulatore il byte al fine di verificare se è uguale a 0 oppure no. Se il valore all'interno dell'accumulatore è 0, allora si ritornerà a controllare la tastiera ripetendo questa subroutine. In caso contrario, si ricorrerà all'uso di un'altra subroutine per la stampa su video del carattere al tasto premuto. Fino a qui, tutto bene.

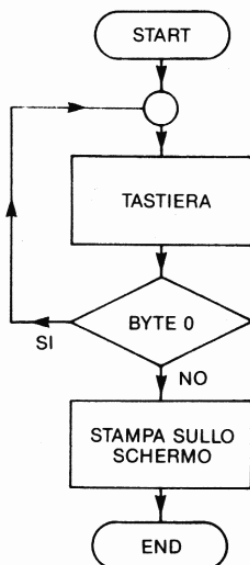



Figura 6.9 - Il diagramma per un programma di stampa caratteri

```
CICLO      JRS $E112; tastiera
           CICLO BEQ
           JRS $E10C; video
```

Figura 6.10 - La versione del programma in linguaggio assembler

La figura 6.10 mostra la versione in linguaggio assembler di questo diagramma di flusso, con la marcatura 'CICLO' usata di nuovo per indicare l'indirizzo al quale il programma deve ritornare se il byte nell'accumulatore è uguale a zero.

La figura 6.11 mostra il programma che utilizza un set di istruzioni

```
10 POKE 56,159:A=40704
20 FORN=1TO9:READ D%
30 POKE A+N,D%:NEXT
40 SYS 40705
100 DATA32,18,225,240,251,32,12,225,96
```

Figura 6.11 - Il programma BASIC che utilizza POKE

POKE in linguaggio BASIC. Quando si esegue questo programma, ogni carattere premuto apparirà sullo schermo al posto del cursore. Ciò non sembra diverso da quando si preme un tasto in qualsiasi altro momento, perciò come capiremo che il programma funziona? Facile: eseguitelo e premete un qualsiasi tasto. Quindi premete il tasto RETURN. Vedrete apparire di nuovo la lettera seguita, nella linea successiva, dal messaggio READY.

Non compaiono messaggi di errore come invece accadrebbe se premeste un tasto qualsiasi seguito dal tasto RETURN.

Perché la lettera viene duplicata? Perché la routine di immissione la registra anche all'interno del buffer di tastiera e, premendo RETURN, si occupa di questo carattere. Il messaggio di errore "?SYNTAX ERROR" manca perché abbiamo cortocircuitato la routine.

Abbiamo parlato di molte cose nuove in questo programma, perciò è giunto il momento di passare oltre con prudenza e assicurarci di avere appreso quanto detto in precedenza, prima di approfondire oltre l'argomento.

Altri cicli

Il ciclo che abbiamo appena provato è classificato come "ciclo di tenuta". Il suo compito è mantenere in attività un ciclo fino a quando non accade un determinato evento.

Ora consideriamo un altro tipo di ciclo chiamato "ciclo contatore". Riveste una duplice importanza: è il modo di ritardare l'esecuzione di un programma in linguaggio macchina e, allo stesso tempo, è il modo di dimostrare la rapidità d'esecuzione del linguaggio macchina.

Il tipo di ciclo usato maggiormente nella programmazione in BASIC è il FOR..NEXT. Esso utilizza una variabile di conteggio per sapere quante volte è stato eseguito il ciclo, confrontandolo con il valore massimo che può assumere e che avete messo al suo interno. Tuttavia, l'operazione svolta dal ciclo FOR...NEXT può essere simulata in BASIC senza ricorrere al suo uso. Il metodo è illustrato nella figura 6.12.

```
10 C=0:ND=10
20 PRINT"AZIONE ";C
30 C=C+1
40 IF C<=ND THEN 20
50 PRINT"FINITO"
```

Figura 6.12 - Un semplice ciclo in BASIC che fornisce l'operazione dell'istruzione FOR...NEXT

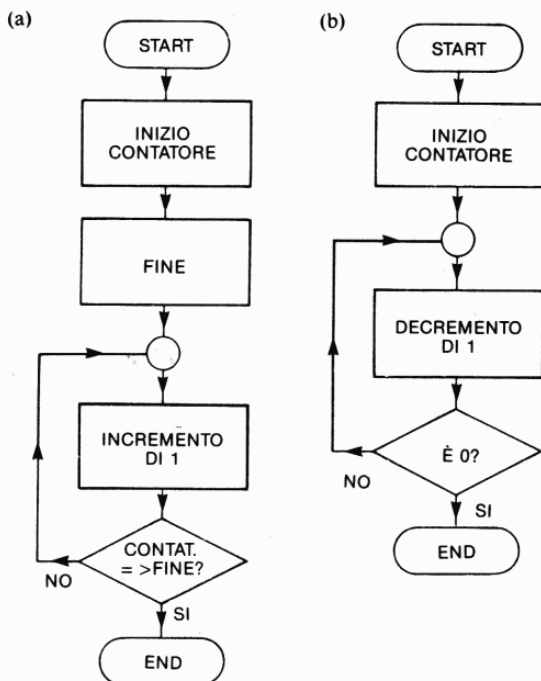



Figura 6.13 - I diagrammi per i cicli. (a) incremento del contatore; (b) decremento del contatore, che è più semplice

Il numero contatore è C, mentre il suo limite è ND. A programma ultimato, il valore di C è 11, come il valore del contatore alla fine dell'esecuzione del ciclo FOR N=1 TO 10. La cosa da farsi successivamente per questo tipo di programma è di guardare il diagramma che abbiamo illustrato nella figura 6.13. Questo modo di formare un ciclo contatore è quello usato in linguaggio macchina. Potremmo scrivere qualcosa in linguaggio macchina che compia la stessa funzione ma, come al solito, dovremmo pensare più a lungo a come portare a termine questo compito.

Per rappresentare un valore, non vengono usati nomi di variabili in linguaggio macchina. Si deve decidere dove registrare un numero, e in quale registro eseguire l'operazione di decremento. Il passo della decisione è più facile. Questa volta si può fare uso di un test BNE per fare ripetere il ciclo fino a quando il contenuto del registro controllato non sia diverso da 0. Nel caso vi chiedeste quale registro state controllando, la risposta è che è sempre quello usato poco prima del test BNE (o di qualsiasi altro).

(a)

```
              LDX #$FF  A2 FF
CICLO         DEX       CA
              BNE CICLO D0 FD
              RTS
```

(b)

```
10 POKE 56,159:A=40704
20 FORN=1TO6:READ D%
30 POKE A+N,D%:NEXT
40 PRINT"PARTENZA"
50 SYS 40705
60 PRINT"ARRIVO"
100 DATA162,255,202,208,253,96
```

Figura 6.14 - (a): un ciclo contatore in linguaggio macchina; (b) programma BASIC con POKE

La figura 6.14 (a) mostra cosa si ottiene con un programma in linguaggio macchina del genere. Il registro usato è il registro X, invece dell'accumulatore, perché (insieme al registro Y) è più adatto per le operazioni di conteggio. Ciò che lo rende più adatto è la presenza delle istruzioni di incremento e di decremento. INX e DEX, rispettivamente, incrementano e decrementano il registro X. Le istruzioni INY e DEY svolgono le stesse operazioni sul registro Y. Non esiste una coppia di istruzioni corrispondenti per l'accumulatore e, se si vogliono eseguire operazioni di conteggio usando l'accumulatore, dobbiamo affrontarle in maniera lievemente differente. In questo esempio, quindi, il registro viene caricato del valore $FF, cioè 255 in decimale. Questo è il numero più grande che possa essere registrato all'interno di un registro ad otto bit. Dopo avere caricato l'accumulatore, lo decrementiamo e sigliamo questo ciclo col marcatore CICLO nel punto in cui vogliamo tornare se il valore contenuto nel registro non è uguale a zero.

Il tasto viene eseguito da BNE (salto se non uguale a zero) perché vogliamo che il programma ripeta l'operazione di decremento fino a quando il contenuto del registro X non sia azzerato. Il programma BASIC che registra i byte nella memoria ed esegue il programma è illustrato nella figura 6.14 (b). Ora, quando si esegue quest'ultimo, non avvertirete un lungo lasso di tempo intercorrente tra la stampa di "INIZIO" e quella di "FINE" e ciò non perché non è accaduto nulla, ma per la rapidità

con cui viene effettuato il conto alla rovescia. Se provate la versione in BASIC di questo programma, noterete che tra le due stampe il periodo di tempo che intercorre è maggiore.

```
10 A=255:PRINT "INIZIO"
20 A=A—1
30 IF A < > 0THEN 20
40 PRINT "STOP"
```

Tuttavia, la differenza non rispecchia il rapporto delle due velocità, perché viene impiegato molto tempo nelle operazioni di stampa della parte BASIC di ogni programma. Per rendersi conto dei grandi vantaggi che può dare il linguaggio macchina in termini di velocità, bisogna lavorare su un numero maggiore di numeri. Per fare questo ci sono vari modi e quello che osserveremo utilizza due cicli. Avrete di certo incontrato dei cicli modificati in BASIC. Il concetto consiste nella presenza di un ciclo interno ed uno esterno. In ogni passo del ciclo esterno, il ciclo interno viene eseguito per intero. Ciò permette di creare dei ritardi molto più lunghi, eseguendo un conteggio all'interno dell'altro. Supponiamo di avere le linee in BASIC:

```
10 X=100: "START"
20 X=X—1
30 Y=255
40 Y=Y—1
50 IF Y< >0THEN 40
60 IF X< >0THEN 20
70 "STOP"
```

esse eseguirebbero un conto alla rovescia di Y da 255 a 0 ogni volta che X viene decrementato. Provatelo e misuratelo, non c'è bisogno di un cronometro a scatto, qualsiasi orologio che abbia le lancette per i minuti andrà bene!

Al contrario, guardiamo come gli stessi numeri verrebbero trattati da un conto alla rovescia in linguaggio macchina. La figura 6.15 (a) mostra la versione in linguaggio macchina. Nel registro X viene caricato il valore $FF e in quello Y il valore $64 (100 decimale). Questa seconda istruzione viene marcata "CICLO 2". Qui compare DEY, in modo tale che il registro Y viene decrementato, e ciò viene marcato "CICLO 1". Il test BNE torna nel punto del CICLO 1 fino a quando il registro X non ha raggiunto il valore 0. Dopo di che, viene decrementato il registro di X, e quindi verificato. Noterete che questa versione non è la stessa di quella in BASIC. Nelle operazioni di verifica e decremento in linguaggio macchina, il decremento avviene prima della verifica, altrimenti il registro

(a)

```
                LDX #$FF      A2 FF
CICLO 2:        LDY #$64      A0 64
CICLO 1:        DEY           88
                BNE CICLO 1   D0 FD
                DEX           CA
                BNE CICLO 2   D0 F8
                RTS
```

(b)

```
10 POKE 56,159:A=40704
20 FORN=1TO11:READ D%
30 POKE A+N,D%:NEXT
40 PRINT"PARTENZA"
50 SYS 40705
60 PRINT"ARRIVO"
100 DATA162,255,160,100,136,208,253,202,208,248,96
```

Figura 6.15 - (a) Linguaggio assemblatore per un conteggio a doppio ciclo. (b) Il programma BASIC con POKE

verificato potrebbe non essere quello giusto. Se il registro X non ha raggiunto lo zero, i cicli di programma vengono ripetuti al CICLO 2, per terminare il registro Y e svolgere di nuovo il ciclo interno.

Quando eseguirete questo programma noterete che sarà troppo rapido per essere seguito! Ciò permette di mostrare le differenze di velocità tra il BASIC e il linguaggio macchina. Se non siete particolarmente convinti che il conteggio sia stato eseguito, allora sostituite il valore del contatore esterno da $64 in $FF (sostituite 100 con 255 alla linea 100 nel programma BASIC). Ciò renderà più vistoso il ritardo.

Le istruzioni INC e DEC dell'accumulatore

Precedentemente ho affermato che non esistono istruzioni INC e DEC. Ciò non significa che non si possa fare uso dell'accumulatore per effettuare delle operazioni di conteggio, ma che è solo meno adatto dei registri X e Y. Tuttavia, supponiamo di dover effettuare un conteggio nell'accumulatore; potremmo fare uso di SBC #1, che sottrae 1 dal contenuto dell'accumulatore. Arriveremmo ad un programma per il conto al-

(a)

```
          CLC        18
          LDA #$FF   A9 FF
CICLO:    SBC #1     E9 01
          BNE FC     D0 FC
          RTS        60
```

(b)

```
10 POKE 56,159:A=40704
20 FORN=1TO8:READ D%
30 POKE A+N,D%
40 PRINT"PARTENZA"
50 SYS 40705
60 PRINT"ARRIVO"
100 DATA24,169,255,233,1,208,252,96
```

Figura 6.16 - Come portare a termine un conto alla rovescia usando un byte nell'accumulatore. (a) In linguaggio macchina. (b) In BASIC usando l'istruzione POKE

la rovescia del tipo mostrato nella figura 6.16 (a). Inizieremo facendo uso di CLC, istruzione che azzera il bit carry.

Ciò è necessario se il bit carry è settato a 1; allora la prima operazione SBC sottrarrà 2 invece di 1. Le restanti operazioni SBC saranno normali e non farebbe molta differenza, in un ritardo come questo, avere un decremento in più. In alcuni tipi di operazione di decremento, tuttavia, come i byte di conteggio, il sottrarre 2 anziché 1 potrebbe risultare catastrofico. Per questa ragione, è buona abitudine azzerare il bit di carry prima di eseguire una sottrazione di questo tipo.

Il passo SBC# 1 è di decremento, e il ciclo è molto simile a quello del registro X di prima. La versione in BASIC che utilizza le istruzioni POKE appare nella figura 6.16 (b).

Il microprocessore 6502 non si limita a decrementare i numeri contenuti nei registri, tuttavia l'operazione DEC può essere applicata a qualsiasi indirizzo della memoria e perciò si possono scrivere programmi di conto alla rovescia che usano quanti byte si desideri. Gli indirizzi più convenienti per registrare i byte di conteggio sono quelli a pagina 0 (da 0 a 255), ma quando il 6502 è installato sul Commodore 64, dovete usare molta prudenza perché il computer usa molti indirizzi di pagina 0 per

i suoi scopi. Mettere perciò i byte in alcuni altri indirizzi creerà la distruzione delle operazioni della macchina.

Tramite l'istruzione PEEK si può constatare che nelle pagine 0 i numeri degli indirizzi dal 150 al 177 non vengono utilizzati durante semplici programmi e perciò potremo utilizzarli per grandi conteggi. La figura 6.17 (a) mostra la versione in linguaggio macchina. I tre indirizzi già usati sono $96, $97, $98 (150, 151 e 152 decimali). Il programma inizia caricando $FF nell'accumulatore, per poi memorizzare questo numero in tutti e tre gli indirizzi di pagina 0. Poiché sono indirizzi di pagina 0, si può fare uso dell'indirizzamento a pagina 0. Il programma non fa altro uso dell'accumulatore, perciò il numero $FF vi rimane per tutto il tempo. Ciò ci permette di registrare l'accumulatore in ciascun indirizzo della memoria senza ricaricarlo.

La tecnica per il conteggio alla rovescia è molto simile alla precedente, tranne per il fatto che fa uso di 3 cicli. Potreste tentare di tracciarvi un diagramma per vedere come vengono sistemati. Il risultato ottenuto

(a)

```
           LDA  #$FF
           STA  $6A
CICLO 1:   STA  $6B
CICLO 2:   STA  $6C
CICLO 3:   DEC  $6C
           BNE  CICLO 3
           DEC  $6B
           BNE  CICLO 2
           DEC  $6A
           BNE  CICLO 1
           RTS
```

(b)

```
10 POKE 56,159:A=40704
20 FORN=1TO21:READ D%
30 POKE A+N,D%:NEXT
40 PRINT"PARTENZA"
50 SYS 40705
60 PRINT"ARRIVO"
90 DATA169,255,133,150,133,151,133,152,198,152,208
110 DATA252,198,151,208,246,198,150,208,240,96
```

Figura 6.17 - Un contatore più lungo che utilizza gli indirizzi di pagina zero per la memorizzazione. (a) linguaggio Assembly; (b) programma BASIC con POKE

è un conto alla rovescia dal numero 16.777.215 (decimale) a 0. Come vi aspetterete, ciò comporta tempi più lunghi, dell'ordine di vari minuti, in confronto ad un conto alla rovescia a due registri.

La versione BASIC compare nella figura 6.17 (b). Se usate un cronometro per misurare l'intervallo di tempo che intercorre tra la stampa dei messaggi "INIZIO" e "FINE", e dividete il numero ottenuto per il numero mostrato sopra, avrete l'idea di quanto tempo (in media) impieghi un ciclo ad essere eseguito. Non provate a fare il conteggio in un programma BASIC. Non basterebbe una vita!

Capitolo 7

Entrate, uscite e cicli

Cicli sullo schermo

Di tutti i programmi a ciclo di cui possiamo fare uso in linguaggio macchina, i cicli che implicano gli indirizzi della memoria dello schermo sono tra i più utili.

In precedenza abbiamo visto che tramite l'istruzione POKE si può maneggiare la memoria dello schermo per fare apparire su di esso quello

Figura 7.1 - Un diagramma per riempire lo schermo con un carattere

che si desidera. Le tecniche usate con l'istruzione POKE in BASIC servono anche per il linguaggio macchina, e le differenze principali sono che il linguaggio macchina comporta una maggiore velocità, ma anche più lavoro da parte vostra!

Per iniziare, date uno sguardo alla figura 7.1. In essa appare un diagramma di flusso per un programma che riempie in parte lo schermo, usando un carattere. Si carica un registro (è di norma avvalersi dell'accumulatore) con un numero di codice per un carattere, poi lo si immagazzina al primo indirizzo dello schermo ($400). Quindi si procede a incrementare questo indirizzo, a registrarlo di nuovo nell'accumulatore e a ripetere questo procedimento fino a quando ne abbiamo bisogno, fino al raggiungimento dell'ultimo indirizzo. Il diagramma di flusso mostra quello che si deve fare, ma è necessario sapere come eseguirlo sul 6502. Questo è il tipo di programma che può fare uso dell'indirizzamento indicizzato, perciò inizieremo col ricordare cosa esso comporti.

Esistono due indirizzi indice, X e Y, ciascuno dei quali può registrare un numero di un solo byte. Quando si esegue un caricamento o registrazione indicizzati (o qualsiasi altra operazione che copia un byte dalla memoria), il numero contenuto nell'INDEX REGISTER (registro indice) viene aggiunto all'indirizzo che abbiamo specificato. Ne risulta un nuovo indirizzo, il quale viene usato per i caricamenti e le memorizzazioni. Se, ad esempio, abbiamo 2 memorizzato nel registro X e specifichiamo un caricamento come:

```
STA 1024,X
```

(con numeri decimali), significa che l'indirizzo uscito sarà 1024+2=1026, e che il carattere contenuto nell'accumulatore verrà registrato nell'indirizzo 1026. Una delle funzioni che lo rendono così utile è che possiede le istruzioni INX e DEX che incrementeranno e decrementeranno rispettivamente il numero all'interno del registro dell'indice X. Ci sono anche i comandi corrispondenti (INY e DEY) per il registro Y.

Passando alla versione in linguaggio assembler, che viene mostrata insieme al programma BASIC nella figura 7.2, il primo passo è semplice: caricare nell'accumulatore il valore 124. Questo è il codice ASCII che riproduce il simbolo grafico a fermo di scacchiera. Ovviamente, potreste provare qualsiasi altro codice che desiderate. Quindi carichiamo il registro X del valore 0.

Il prossimo passo è il ciclo. Iniziamolo memorizzando il byte dell'accumulatore nell'indirizzo fornito da 1024 (decimale) più il byte del registro X, per poi incrementare il registro X. In linguaggio assembler, questi passi vengono scritti:

STA 1024, X
INX

Il passo successivo è il confronto tra il contenuto del registro X e lo 0, tramite BNE. Dato che il registro X è partito da 0 ed è stato appena incrementato a 1, il confronto non dà zero, così l'istruzione BNE fa sì che si ripeta il ciclo e che il carattere venga immagazzinato in un altro indirizzo. Quando il numero nel registro X è uguale a 255 (decimale), l'incremento successivo azzera il contenuto del registro X. Ciò accade perché il registro può contenere al massimo un byte.

A questo punto il programma esce dal ciclo BNE per tornare in ambiente BASIC.

Questo programma esegue solo in parte quello che vorremmo. Pone un carattere in 256 indirizzi dello schermo, ma lo schermo è formato da 1000 locazioni di memoria. Ciò che ci ha limitato in questo caso è la dimensione del registro X: un solo byte. Essa non permette di ottenere cicli che trattino più di 256 byte per caricamenti e memorizzazioni. Per ora ci accontenteremo dei risultati ottenuti, più avanti vedremo come aggirare questa limitazione.

Il passaggio da linguaggio assembler a linguaggio macchina è abbastanza facile. Quando i byte del linguaggio macchina sono stati scritti (controllate il byte di scostamento che segue BNE), anche il programma

(a)

```
         LDA #124      ;124 decimale
         LDX #0        ;X comincia da zero
CICLO    STA 1024,X    ;registra 1024+X
         INX           ;incrementa X
         BNE CICLO     ;fino alla fine del conto
         RTS           ;ritorna al BASIC
```

(b)

```
10 POKE 56,159:A=40704
20 FOR N=1 TO 11:READ D%
30 POKE A+N,D%:NEXT
35 POKE 53281,3
40 SYS 40705
100 DATA169,124,162,0,157,0,4,232,208,250,96
```

Figura 7.2 - (a) Il programma in linguaggio Assembly. (b) Il programma BASIC

BASIC che registra i byte all'interno della memoria può essere scritto (fig. 7.2 (b)).

Non dovreste metterci molto. A parte il valore di N e la linea DATA, è quasi lo stesso tempo impiegato nei programmi precedenti. Dobbiamo tuttavia aggiungere il passo POKE53281,3 per assicurare che gli effetti del programma siano evidenti. Alla fine dell'esecuzione, si devono usare i tasti STOP e RESTORE per far sì che lo schermo ritorni nelle condizioni normali. Quando il programma viene eseguito avviene un ritardo mentre il BASIC registra i numeri nella memoria, poi il linguaggio macchina dimostra la sua solita rapidità d'esecuzione.

Altri metodi

La stesura di un programma che riempia l'intero schermo non è poi così facile a causa del limite delle dimensioni del registro indice. Ci sono vari modi per eseguire l'operazione di cui abbiamo bisogno, ma il metodo più semplice fa uso di quello che viene chiamato INDIRIZZAMENTO INDIRETTO. Esso è stato brevemente descritto nel capitolo 4 e ora lo riprendiamo, dando uno sguardo più attento a uno dei due metodi che il 6502 usa.

I due metodi di indirizzamento indiretto del 6502 sono spesso conosciuti come "metodo indiretto indicizzato" e "metodo indicizzato indiretto". Per evitare di fare confusione, in questo libro mi atterrò alla più semplice denominazione: X indiretto e Y indiretto, e questo perché un metodo fa uso del registro indice X, mentre l'altro del registro indice Y. Quello che useremo per il programma del ciclo è il metodo ad Y indiretto.

Il modo in cui opera l'indirizzamento a Y è il seguente: il primo indirizzo che vogliamo usare, nel nostro caso $0400, viene registrato utilizzando due byte in due locazioni consecutive nella memoria. Come sempre, i byte vengono memorizzati seguendo l'ordine low-high (basso-alto) e devono essere nella pagina 0 della memoria. Viene messo un numero anche nel registro dell'indice Y. Ora, quando viene eseguita un'operazione come un caricamento o una memorizzazione, viene usata la formula a Y indiretto. Nel linguaggio assemblatore, una memorizzazione a Y indiretto verrebbe scritta nella forma:

STA indirizzo, Y

Ciò che produce è piuttosto complicato, a prima vista. Il byte basso dell'indirizzo viene copiato dalla memoria, e ad esso viene aggiunto il contenuto del registro Y. Se da questa somma scaturisce un riporto, esso viene aggiunto nel byte alto. I due nuovi byte dell'indirizzo vengono

quindi usati (in questo esempio) per l'operazione di memorizzazione.

Supponiamo, ad esempio, di aver immagazzinato nella memoria di pagina 0 il primo indirizzo della memoria dello schermo $0400, usando gli indirizzi 150 e 151 (dec.). Usare l'ordine di memorizzazione LO-HI (basso-alto) significa che l'indirizzo 150 conterrà $00 e 151 $04. Ora, se il registro Y contiene il numero $26 (esadec.), allora l'effetto di:

STA(150), Y

sarà di aggiungere $26 al numero memorizzato nella locazione 150 (che era 0), e così formare l'indirizzo $0426. Questo è l'indirizzo in cui verrà copiato il byte contenuto nell'accumulatore.

(a)

```
        LDA  #124
        LDX  #0
        STX  150
        LDX  #4
        STX  151
        LDY  #0
CICLO:  STA  (150),Y
        INY
        BNE  CICLO
        INC  151
        LDX  151
        CPX  #8
        BNE  CICLO
        RTS
```

(b)

```
10 POKE 56,159:A=40704
20 FOR N=1TO26:READ D%
30 POKE A+N,D%:NEXT
35 POKE 53281,3
40 SYS 40705
50 GOTO 50
90 DATA169,124,162,0,134,150,162,4,134
93 DATA151,160,0,145,150,200,208,251
95 DATA230,151,166,151,224,8,208,243,96
```

Figura 7.3 - Come riempire tutto lo schermo; anche in questo programma è presente una imperfezione, che in questo caso non causerà alcun problema. (a) In Assembler, (b) in BASIC

La figura 7.3 (a) mostra la versione completa in linguaggio assembler. I primi 6 passi mettono gli esatti valori nei registri e nella memoria. Questa operazione non viene eseguita nel più efficiente dei modi, ma è facile. Le cose iniziano a complicarsi al passo 7, all'inizio del ciclo. La causa del problema è la necessità di eseguire l'incremento degli indirizzi della memoria dello schermo per un migliaio di volte. Daremo uno sguardo a questa parte del programma in maniera dettagliata.

All'inizio del ciclo, il numero nel registro Y è 0 e l'indirizzo base per l'operazione STA è contenuto negli indirizzi 150 e 151 (dec.) della pagina 0. L'indirizzo 150 contiene il numero 0, mentre nel 151 si trova $04; questi due byte formano il primo indirizzo della memoria dello schermo ($0400), cioè 1024 (dec.). Quando viene usato il passo STA(150), Y per la prima volta, allora il byte per l'accumulatore verrà copiato nell'indirizzo $0400. Quindi il passo INY incrementa il numero nel registro Y da 0 a 1. Quando si giunge al BNE LOOP, il programma salterà indietro per ripetere il ciclo a causa del fatto che il registro Y contiene 1. Questo ciclo permette l'uso di indirizzi consecutivi della memoria dello schermo. Di nuovo, ciò si ripete fino a quando il registro Y passa da 255 (dec.) a 0.

In questo modo si esce dal ciclo. Dato che abbiamo iniziato ad indirizzare la locazione con 0, non c'è stata alcuna aggiunta automatica al numero nell'indirizzo 151, e dobbiamo assolvere questo compito da soli. A questo punto ho preso una scorciatoia di cui è necessario conoscere la struttura per capire il motivo della sua efficacia.

Alla fine del primo ciclo, viene incrementato il numero all'indirizzo 151. Esso seleziona automaticamente il successivo quarto dello schermo se si ripete il primo ciclo. Tuttavia, dobbiamo essere in grado di fermare il programma, perché non vogliamo che il byte 124 venga copiato in ogni parte della memoria — sconvolgerebbe il nostro programma. Per formare questa operazione viene copiato il numero all'indirizzo 151 nel registro X, e messo a confronto col numero 8. Se non è uguale a 8, il secondo BNE ritorna al primo ciclo per riempire un'altra parte di memoria.

Quando il numero 151 raggiunge 8, il programma termina. La figura 7.3(b) mostra il programma in linguaggio BASIC che registra i byte nella memoria e gestisce il linguaggio macchina.

Sebbene riempia tutto lo schermo, non può essere sempre usato. Il motivo sta nel fatto che gli indirizzi in realtà vanno da $0400 (1024 dec.) a $07E7 (2023 dec.). Abbiamo usato gli indirizzi da $0400 a $07FF e abbiamo riempito anche gli indirizzi che vanno da $07E7 a $07FF che è la parte della memoria riservata ai puntatori degli SPRITE. Se il vostro programma non utilizza gli sprite, non sorge alcun problema; se invece avete intenzione di usarli e avete bisogno di questa parte della memoria,

potrete agire su di essa in seguito. Se, tuttavia, volete memorizzare dei dati sprite in questa zona della memoria, e poi eseguire il programma in linguaggio macchina, dovrete poter fermare il riempimento dello schermo dopo l'indirizzo $07E8. Questa operazione è qualcosa di più del semplice passo CPX8 usato in questo esempio. È necessario usare il passo CPX7 al posto di CPX8. Seguendo il ciclo BNE, dovrete usare LDX 150 e CPX $E8 per verificare il byte più basso. Questo è seguito da un altro passo per il BNE LOOP. Il risultato di tali operazioni sarà di fermare il programma quando è arrivato all'indirizzo $07E8, risparmiando così i vostri sprite (da una fine peggiore della morte).

Programmi più grandi

Il programma nella figura 7.3 può essere modificato in un modo interessante. Supponiamo di essere partiti con l'accumulatore azzerato, e di averlo incrementato ogni volta che il ciclo si ripeteva. In tal modo possiamo riprodurre sullo schermo ogni carattere rappresentato dai codici a partire dallo 0 fino a 255 (dec.). Siccome l'accumulatore è un registro a un solo byte, il numero 255 è il più grande che può contenere, per cui incrementandolo non si farà altro che ripartire dacapo e cioè da 0. La fig. 7.4 mostra il diagramma di flusso di questa operazione, mentre la fig. 7.5 mostra i programmi nelle versioni in linguaggio macchina e in BASIC.

In questi programmi non c'è nulla che possa procurarvi dei seri grattacapi, perché l'unica differenza tra questi e il programma della figura 7.3 sta nell'incremento dell'accumulatore. Come prima, dobbiamo azzerare il bit carry all'inizio del programma. Con il passo ADC#1 nel ciclo, aggiungerete 1 all'accumulatore per ogni ciclo compiuto. Provatelo, e vedrete apparire l'intera gamma di caratteri dello STATO TESTO. Questo è un buon esempio di come un semplice programma possa essere applicato in modo tale da poter compiere molte più cose, che non nella versione originale. È un punto importante, perché molte programmazioni in linguaggio macchina sono di questo tipo. Se terrete conto di tutti i programmi in linguaggio macchina da voi usati, e di quanto hanno fatto, troverete che questa "biblioteca" è un patrimonio preziosissimo. Molto spesso noterete che si può ottenere qualsiasi nuovo programma abbiate intenzione di scrivere modificando e/o cambiando vecchie subroutine che già conoscete. Un altro grosso vantaggio è che ci si può affidare ad una subroutine già usata in passato poiché è già stata sufficientemente verificata.

Prima di terminare questo paragrafo, diremo cosa accade quando si

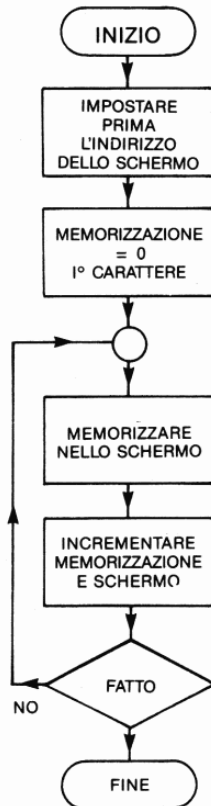



Figura 7.4 - Diagramma di flusso per stampare l'intero set dei caratteri

aggiunge 1 a 255 (decimale) nell'accumulatore. Cosa accadrà al bit carry? (di riporto). È necessario l'uso di CLC nel ciclo per neutralizzare gli effetti? Vedete se riuscite ad individuarne l'effetto e a perfezionarlo.

Salvatelo!

A questo punto, quando i programmi tendono a diventare abbastanza lunghi e cominciamo a svolgere compiti interessanti, giunge il momento di conoscere le tecniche per salvare i programmi in linguaggio macchina su nastro. In ogni caso non siete obbligati a farlo. I programmi in linguaggio macchina sono tutti stati scritti, fino ad ora, utilizzando il linguaggio BASIC e l'istruzione POKE per registrare i numeri all'in-

(a)

```
           CLC
           LDA   #0
           TAX
           STX   150
CICLO:     LDX   #4
           STX   151
           TAY
           STA   (150),Y
           ADC   #1
           INY
           BNE   CICLO
           INC   151
           LDX   151
           CPX   #8
           BNE   CICLO
           RTS
```

(b)

```
10 POKE 56,159:A=40704
20 FOR N=1TO27:READ D%
30 POKE A+N,D%:NEXT
35 POKE 53281,3
40 SYS 40705
50 GOTO 50
90 DATA24,169,0,170,134,150,162,4,134
93 DATA151,168,145,150,105,1,200,208,249
95 DATA230,151,166,151,224,8,208,241,96
```

Figura 7.5 - (a) La versione in linguaggio macchina e (b) il programma in BASIC ottenuti dal programma della fig. 7.4

terno della memoria. Naturalmente possiamo salvare questo programma in BASIC, ed esso creerà il codice macchina per voi in qualsiasi momento lo vogliate.

Il modo migliore di caricare e salvare dei programmi in linguaggio macchina è quello di ricorrere all'uso dei programmi "misti", cioè programmi in cui le istruzioni vengono date in linguaggio macchina e in BASIC. Alle volte, tuttavia, dovrete poter contare su una quantità di memoria il più grande possibile. In questo caso, anche una piccola parte di programma scritta in BASIC sarebbe la benvenuta come lo è un elefante all'interno di una capsula spaziale. Potreste anche voler scrivere dei programmi misti su cassetta in modo tale da rendere difficile la loro copia-

tura. In entrambi i casi, potreste voler registrare direttamente i byte contenuti in memoria.

I normali comandi BASIC del Commodore 64 possono soddisfare queste esigenze ma, come vedremo più avanti, in maniera abbastanza tortuosa. Diversamente da molte altre macchine, il C64 non possiede comandi speciali in BASIC per salvare o caricare programmi in linguaggio macchina. Quando utilizzate un programma supervisore in linguaggio macchina scritto per il C64, troverete che di solito esso include delle routine per salvare e caricare il programma in linguaggio macchina. Questo è un pregio se fate uso del programma supervisore in continuazione, ma sarebbe utile avere anche un altro metodo di caricamento e salvataggio. Fortunatamente, i comandi BASIC SAVE e LOAD possono venir usati per svolgere questa funzione.

Quando viene usato il comando SAVE, il C64 salva un programma in linguaggio macchina. Ciò significa che tutti i byte a partire dall'indirizzo 2049 fino all'ultimo del BASIC vengono salvati. I primi due byte ad essere salvati, infatti, sono sempre quelli che forniscono l'indirizzo di termine del programma in BASIC.

Il linguaggio macchina ottiene questi due byte dalla pagina zero della memoria. L'inizio del BASIC, da quello che sapete finora, è conservato agli indirizzi 43 e 44. Il termine del BASIC, che è lo stesso dell'inizio della lista delle variabili, viene conservato agli indirizzi 45 e 46. Se variano i numeri a questi indirizzi saremo in grado di controllare il salvataggio o il caricamento in ogni altra parte della memoria.

```
10 POKE 56,159:A=40704
20 FOR N=1TO100
30 POKEA+N,N:NEXT
35 POKE 53281,3
40 POKE 43,255:POKE 44,158
50 POKE 45,101:POKE 46,159
60 SAVE"MC"
70 POKE 43,1:POKE 44,8
80 POKE 45,3:POKE 46,8
```

Figura 7.6 - Programma che registra valori all'interno della memoria salvandoli su nastro. Mostra come i programmi in linguaggio macchina possano essere salvati o caricati

La figura 7.6 mostra un programma in BASIC che registra dei numeri nella memoria, salvandoli poi su nastro. Naturalmente, avremmo potuto fare uso di un programma in solo linguaggio macchina, ma la sequenza dei numeri è facile da ricordare. Al solito, la memoria viene allocata in modo tale che i cento numeri non corrano il rischio di essere alterati dalla

macchina. Le linee 40 e 50 sono necessarie per resettare i valori all'interno della pagina zero. Al posto del solito numero di inizio BASIC nelle locazioni 43 e 44, mettiamo un indirizzo che preceda di due byte quello d'inizio del set dei cento numeri. Questi due byte sono importanti, poiché se si usasse l'indirizzo di partenza vero, il sistema operativo sostituirebbe i primi due numeri della serie dei cento con i due byte dell'indirizzo di fine programma. Quindi registriamo il valore dell'ultimo indirizzo del programma agli indirizzi 45 e 46, dopodiché possiamo usare il comando SAVE normalmente. Esso darà il solito messaggio, mentre le linee 70 e 80 ripristineranno i valori normali nelle locazioni di pagina 0.

Ora non resta che vedere se il tutto funziona a dovere. Eseguite (RUN) il programma usando una cassetta vergine, premete i tasti RECORD e PLAY, quindi attendete la fine del programma. Ora riavvolgete il nastro della cassetta sul quale dovrebbero essere presenti tutti i numeri registrati precedentemente nella memoria, quindi spegnete il C64. Nel momento in cui lo riaccenderete tutte le locazioni verranno azzerate, perdendo così i numeri memorizzati all'interno della memoria. Per poter usare la cassetta di nuovo, è necessario azzerare ancora la pagina zero della memoria, caricare, quindi ritrascrivere i valori conservati nella pagina 0. Queste operazioni non possono essere eseguite contemporaneamente poiché l'operazione di caricamento non viene attivata se sono presenti delle linee di programma dopo l'istruzione LOAD.

Per recuperare il programma bisogna per prima cosa assegnare la memoria digitando POKE56,159 e RETURN, quindi cambiare i numeri agli indirizzi 43 e 44, usando:

```
POKE43,255: POKE44,158
```

come prima. Per predisporre i byte nelle appropriate locazioni di memoria, digitate LOAD "MC" dopo di che:

```
POKE43,1: POKE44,8: POKE45,3: POKE46,8.
```

In questo modo verranno ripristinati i valori normali alla pagina zero della memoria. A questo punto potrete controllare la correttezza dell'operazione digitando:

```
FOR N=1 TO 100: ?PEEK(40704+N; " ";: NEXT
```

quindi premete RETURN. Dovrete potere osservare la lista dei numeri che compaiono sullo schermo. Ciò prova come un programma in linguaggio macchina possa essere salvato e caricato utilizzando solo i comandi BASIC.

Non pensate, tuttavia, di poter applicare questa procedura solo coi

programmi in linguaggio macchina. Ogni parte della memoria può essere salvata e ricaricata con questo metodo. Dato che lo schermo è controllato dai byte conservati nelle locazioni che vanno dalla 1024 alla 2023 (dec.), si possono salvare e caricare sullo schermo forme grafiche. Queste vengono però alterate dai messaggi che appaiono durante il caricamento perciò, per ottenere risultati migliori, dovreste staccare il visore dell'output di tastiera, registrando nella locazione 154 il valore 4 (POKE154,4). Per collegare il monitor all'output di tastiera usate POKE154,3. Mentre lo schermo è disattivato, non vedrete apparire nulla digitando POKE154,3. L'effetto sarà normale premendo RETURN e facendo un qualsiasi uso della tastiera, come un listato di programma.

Prendere un messaggio ...

Dopo questo breve interludio sul salvataggio e ricaricamento di programmi in linguaggio macchina, torniamo alla programmazione. Siamo rimasti alla figura 7.5 che scriveva dei caratteri sullo schermo. Ora passeremo alla stampa di cose più interessanti, e in effetti stampare lettere sembra essere un punto di partenza ragionevolmente semplice per questo tipo di programmazione. Cosa si deve fare?

Per cominciare dobbiamo memorizzare alcuni codici ASCII in qualche parte della memoria per le lettere; magari utilizzando una variabile stringa come viene usata in BASIC. Dovremo inoltre essere a conoscenza del primo indirizzo in cui memorizzare la prima lettera, e quante lettere memorizzare a partire da quell'indirizzo. Fatto questo, dovremmo essere in grado di scrivere un ciclo che prelevi un byte dalla memoria di testo per registrarlo nella memoria dello schermo. Avendo già usato i principali tipi di istruzioni di cui abbiamo bisogno per questo tipo di cose — il caricamento e il salvataggio di autoincremento — possiamo passare al lavoro.

Come sempre si parte dal diagramma di flusso, che questa volta presenta maggiori difficoltà nella stesura perché abbiamo bisogno di terminare il ciclo in maniera differente. Si potrebbe contare il numero delle lettere che vogliamo stampare sullo schermo, ma per questa volta prenderemo in esame una tecnica diversa, usando un terminatore. Siete già a conoscenza di questo concetto usato nella programmazione in BASIC.

Il "terminatore" è un byte che il programma è in grado di riconoscere come carattere speciale — ad esempio, uno che non faccia parte del messaggio. Un comodo terminatore adatto a molti scopi è lo 0, perciò lo utilizzeremo. Sorge però il problema di non farlo comparire sullo schermo. Secondo il modo in cui il C64 utilizza i codici numerici, in realtà

lo zero darebbe come risultato la stampa del carattere @ (chiocciolina). Ma noi non vogliamo che esso appaia, perciò dovremo controllare l'accumulatore tra il caricamento del byte dalla memoria e la stampa del corrispondente carattere sullo schermo, rendendo più complessa la stesura del ciclo.

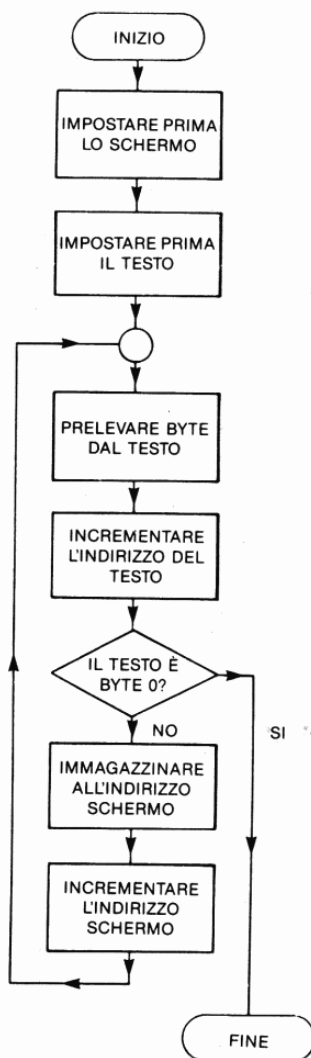



Figura 7.7 - Il diagramma di flusso per scrivere un messaggio sullo schermo

Il diagramma di cui abbiamo bisogno è mostrato nella figura 7.7. Ciò che dobbiamo fare è di memorizzare due locazioni della memoria. Una di queste vi sarà familiare perché farà parte della memoria dello schermo. Questo sarà l'indirizzo del primo carattere che vorremmo fare apparire. Il secondo indirizzo dovrà essere quello di partenza per la memorizzazione di una stringa di byte usati per memorizzare i codici ASCII, che non verrà usata per altri scopi. Per rendere le cose più comprensibili, potremo designare ad esempio SCRN la memoria dello schermo e TXT la parte della memoria in cui vengono registrati i codici. Dopo aver allocato questi indirizzi, caricheremo un codice da TXT, incrementando l'indirizzo di TXT. Quindi controlleremo che esso non sia il nostro terminatore (0) perché se lo fosse, il programma terminerebbe. In caso contrario questo carattere verrebbe registrato nella SCRN, incrementando l'indirizzo di SCRN, per poi ritornare indietro e ricominciare daccapo

(a)

```
             LDX #0
CICLO:       LDA TXT,X    ;preleva il carattere
             CMP #0       ;verifica se il carat-
                           tere è uguale a 0
             BEQ OUT      ;termina il program-
                           ma se il carattere è
                           uguale a 0
             STA SCRN,X   ;registra il carattere
                           sullo schermo
             INX          ;incrementa l'indice
             BNE CICLO    ;fa ripetere il processo
OUT:         RTS          ;ritorna al BASIC
```

(b)

```
10 POKE 56,159:A=40704:B=40863
20 FOR N=1 TO 16:READ D%
30 POKE A+N,D%:NEXT
40 REM DEFINIZIONE POKE MEDIANTE LETTERE
50 FORJ=1 TO 13:READ D%:POKE B+J,D%:NEXT
60 PRINT"□"
70 POKE 53281,3
80 SYS 40705
100 DATA162,0,189,160,159,201,0,240,6
110 DATA157,224,5,232,208,243,96
120 DATA3,15,13,13,15,4,15,18,5,32,54,52,0
```

Figura 7.8 - (a) Il programma in linguaggio macchina della routine di stampa di un messaggio, e (b) il listato in BASIC

lo stesso procedimento. Per fare tutto questo dovremo scrivere un diagramma di flusso in cui sono presenti due salti. Uno di questi è relativo al salto all'indietro, l'altro è relativo al salto verso la fine del programma. Come apparirà il programma in linguaggio assembler?

La risposta appare nella figura 7.8. Esso segue esattamente il diagramma di flusso, ma dovremo prestare particolare attenzione ai passi BEQ e BNE. Durante l'esecuzione del passo BEQ, nell'accumulatore è già stato caricato un dato proveniente dalla memoria TXT, il cui indirizzo di partenza è $9FA0.

Questo indirizzo viene tradotto in decimale nei due byte 160,159; il che equivale all'indirizzo 40864. Ho preso un indirizzo interessante ben distanziato da quelli usati nel programma. L'accumulatore viene caricato usando l'indirizzamento indicizzato di X, e il registro X settato a 0. Come risultato si avrà che il primo caricamento dell'accumulatore avverrà dall'indirizzo 40864, cioè il byte 3, che è il codice interno del C64 per rappresentare la lettera C quando si sta usando questo metodo di stampa sullo schermo.

Il passo CMPO viene scritto dopo il caricamento per poter rivelare il byte 0 alla fine della stampa del messaggio. BEQ (salta se uguale a zero) provvederà a far saltare il programma al passo contenente l'istruzione RTS, saltando i passi compresi tra le istruzioni BEQ e RTS. Se il byte non è 0, viene registrato nell'indirizzo della SCRN, usando di nuovo l'indice X per l'indicizzazione. Per poter saltare indietro si farà, poi, uso dell'istruzione JMP che deve essere seguita da un indirizzo di due byte rendendo più facile l'utilizzazione dell'istruzione BNE. A questo punto del programma, il byte all'interno del registro X non deve mai essere uguale a 0 (a meno che non tentiate di mettere troppe lettere sullo schermo). Come risultato, BNE farà tornare il programma sempre alla posizione LOOP (ciclo). Ciò continuerà fino a quando non si caricherà uno 0 nell'accumulatore, e il passo BEQ OUT obbligherà il programma a tornare in ambiente BASIC.

Spiegato ciò, possiamo tradurlo nella forma di programma BASIC e verificarne l'efficacia. Metteremo i byte all'interno della memoria usando un modo abbastanza semplice, cioè registrandoli in memoria tramite l'istruzione POKE della linea DATA, che è quanto avviene alla linea 50. Le linee da 20 a 30 hanno precedentemente inserito il programma in linguaggio macchina, mentre la linea 80 lo esegue. Durante l'esecuzione vedrete apparire il messaggio sullo schermo.

Adesso tratteremo un punto fastidioso a proposito del collocamento del testo sullo schermo. In tutti i precedenti programmi che fanno uso di testi su video, si è dovuto regolare il colore dello sfondo per rendere visibili i caratteri. In un programma ciclico come questo, si può fare di

meglio. Ciò che faremo sarà di mettere il numero 1 in ogni punto delle lettere nelle locazioni del colore. Se ricordate il BASIC del C64, ricorderete che aggiungendo 54272 (dec.) cioè $D400 a un indirizzo della memoria di testo, si ottiene l'indirizzo corrispondente del colore. Registrando, tramite POKE, si possono ottenere diversi colori in evidenza dei caratteri.

Questo tipo di cose necessita solo di una piccola variazione al programma in linguaggio macchina. La fig. 7.9 mostra la versione in assembler. Dopo il passo STA SCRN,X che posiziona il codice del carattere nella memoria di testo, si hanno i passi LDA #1 e STA COLOR,X. Questi passi immettono il numero 1 nei corretti indirizzi della memoria del colore, purché sia presente il corretto indirizzo di base. L'indirizzo di base per i testi è $05E0, che aggiunto a $0400 dà $D9E0, cioè 55776 decimale. Usando questo metodo per collocare i caratteri sullo schermo, si evita l'uso dell'indirizzo 53281 e la necessità di premere i tasti STOP e RESTORE alla fine del programma. Un altro passo in avanti.

```
            LDX   #0
CICLO:      LDA   TXT,X
            CMP   #0
            BEQ   OUT
            STA   SCRN,X
            LDA   #1
            STA   COLR,X
            INX
            BNE   LOOP (CICLO)
OUT:        RTS
```

Figura 7.9 - Modifica della routine per poter registrare nella memoria del colore; ciò assicura la lettura del testo senza incorrere in un cambiamento del colore di fondo

Come dirigersi fuori dal POKE

Quando ho descritto le funzioni del sistema del computer nel capitolo 1, è stato chiarito il concetto di "PORT". Per quanto riguarda il computer, un PORT è l'insieme di un chip, o più chip, che svolgono le operazioni di INPUT e OUTPUT dei dati. Nella realtà, l'organizzazione dei PORT del C64 è piuttosto complicata. Per nostra fortuna, partendo dall'uso della grafica più avanzata, a meno di non voler fare delle registrazioni insolite su cassetta, non dobbiamo tenere il controllo dei PORT.

Il sistema per il suono e gli effetti speciali per il video, come gli sprite, sono trattati da chip speciali.

In effetti, l'esecuzione di queste operazioni in linguaggio macchina è particolarmente semplice, perché per farlo non abbiamo bisogno di capire il sistema dei PORT. Il C64 usa l'istruzione POKE in BASIC per i comandi del suono. Ciò significa che possiamo usare gli stessi metodi in linguaggio macchina, tramite l'istruzione LDA (immediato) per prendere un byte, e STA per metterlo all'interno della memoria. Ovunque si possa fare qualcosa tramite l'istruzione BASIC POKE, si può usare la combinazione di LDA e STA in linguaggio macchina. In ogni modo ce ne preoccuperemo molto raramente perché possiamo ottenere gli stessi effetti usando il BASIC. Non c'è ragione di fare uso del linguaggio macchina per la gestione del suono, perché la velocità del linguaggio macchina rappresenta uno svantaggio in quanto si è costretti in ogni modo a ricorrere all'uso di cicli di ritardo per ottenere la giusta durata di un suono. Solo nei casi in cui vogliate raggiungere risultati sullo schermo, altrimenti impossibili tramite la normale grafica degli sprite, dovrete avere la padronanza sul "VIDEO INTERFACE CHIP" del C64. Questa non è chiaramente cosa adatta ai principianti del linguaggio macchina, perciò la lasceremo da parte.

Capitolo 8

La ricerca degli errori, il controllo, il MIKRO

La ricerca degli errori

Dopo aver fatto esperienza con alcune parti della programmazione in linguaggio macchina, è giusto parlare degli inconvenienti a cui si va incontro. Uno di questi è il recupero degli errori, cioè la loro ricerca e la loro successiva correzione.

La miglior cosa, anche se è facile a dirsi, è la prevenzione. Controllate il vostro diagramma di flusso per assicurarvi che in esso vengano descritte le operazioni che intendete eseguire. Quando sarete soddisfatti del diagramma, traducetelo nel linguaggio macchina per verificare se esso esegue le operazioni descritte. Se il tutto sembra funzionare bene, controllate che i byte che intendete registrare nella memoria siano i corrispondenti byte delle istruzioni in linguaggio macchina. Dovete poi stare molto attenti ad usare il codice esatto per il metodo di indirizzamento utilizzato. Se controllate ogni passo dello sviluppo di un programma in questo modo, eviterete di commettere molti errori. Se a questo punto il programma non funziona ancora, non scoraggiatevi! A meno che non si tratti di un programma breve, le probabilità di incappare in un errore sono elevate. Capita a tutti, e il fatto di commetterne pochi e di saperli trovare facilmente dipende solo dal livello di esperienza maturato.

Usando un assemblatore, eliminerete una fonte di errori poiché la disattenzione umana fa sì che la conversione delle istruzioni in linguaggio assembler nei codici del linguaggio macchina porti a commettere errori. La ragione di questo sta nella ricerca sulle tavole di conversione dei codici necessari alla traduzione di un programma che può facilmente comportare sviste di ogni genere.

In questo capitolo verranno brevemente descritte le funzioni dell'assembler MIKRO. Sebbene nel momento in cui questo libro è stato scritto esistessero già vari assemblatori per il C64, il MIKRO presenta molti

vantaggi, uno dei quali è che lo si può avere su cartuccia insieme con un programma supervisore chiamato TIM. Una parte di questo capitolo è stata riservata per i programmi supervisori. Se il linguaggio macchina vi ha interessato e se ve la sentite di affrontare qualcosa di più avanzato di ciò che è stato descritto in questo libro, allora un buon assemblatore e un programma supervisore sono essenziali: dovrete essere pronti a dedicare molto tempo a questi programmi. Se, tuttavia, avete semplici intenzione di dilettante e vi piace creare nuovi programmi, allora i metodi di registrazione nella memoria mediante l'istruzione POKE usati fin qui sono perfettamente adatti.

L'uso di tali metodi, tuttavia, significa che ci saranno errori in agguato in ogni angolo del programma. La causa principale di questi errori è la stanchezza. La conversione di un programma in linguaggio assembler in valori esadecimali e il trascriverli sotto forma di linea DATA in un programma BASIC che usi l'istruzione POKE è un compito tedioso e, come in tutti i compiti tediosi, compaiono inevitabilmente degli errori. Come risultato si avranno errori nei metodi di indirizzamento e nella trascrizione di codici numerici. Un'altra causa di errori può venire da salti effettuati all'indirizzo sbagliato. Si possono ottenere numeri inesatti nelle sottrazioni di indirizzi e nella conversione di numeri nella notazione esadecimale (specialmente quando si tratta di numeri negativi).

Altri problemi possono sorgere quando si modifica un programma, aggiungendo un codice tra una istruzione di salto e la sua destinazione, perché è facile dimenticarsi di cambiare il valore del byte di scostamento! Problema, questo, che non sorge qualora si faccia uso di un assemblatore. Un salto errato blocca quasi sempre il computer e per ripristinare il controllo si può ricorrere all'uso dei tasti RUN/STOP e RESTORE; ma purtroppo a volte si causa la perdita del programma (vi siete premuniti registrandolo su cassetta, non è vero?).

Un'altra forma di salto scorretto è fare l'opposto di quello che intendevate fare, usando BEQ al posto di BNE o altre cose del genere. Un'attenzione particolare a quanto il salto opererà per valori diversi di byte dovrebbe eliminare questo problema.

La maggior parte delle difficoltà possono essere eliminate, come ho già detto, tramite un attento controllo, facendo estrema attenzione agli scostamenti dei salti e ai contenuti iniziali dei registri. Un errore molto comune è di fare uso dei registri all'inizio del programma come se contassero 0. Non potendo essere sempre certi che ciò accada, è più sicuro supporre che ogni registro contenga un valore che porti il computer a sbagliare. Detto ciò, e con tutto lo sforzo di volontà del mondo, cosa farete se il programma non funziona ancora?

La risposta non è semplice. Può darsi che il diagramma di flusso non dia i risultati attesi, o forse non lo avete disegnato e a questo punto avete quel che meritate. Può darsi che voi stiate tentando di usare una routine della ROM del C64 e che essa non funzioni nel modo desiderato. Quando avrete abbastanza esperienza del linguaggio macchina da capire programmi più elaborati, potrete fare uso di un assemblatore della ROM (ciò vi permetterà di convertire il codice macchina in linguaggio assembler). Esiste a questo proposito un utile libro intitolato INSIDE THE COMMODORE 64 di Milton Bathurst, pubblicato dalla Datacap in Belgio che è disponibile nelle maggiori librerie. Questo libro possiede l'elenco completo, con relativi commenti, della ROM del Commodore 64. Inoltre si possono trovare molte informazioni sull'uso della RAM e, in particolare, sugli indirizzi di pagina 0. Questo libro è adatto per i vari programmatori in linguaggio macchina del C64.

Tutto quello che posso fare ora è di darvi alcune direttive generali sulla rimozione degli errori da un programma che in apparenza sembra ben strutturato, ma che poi non funziona secondo il piano prestabilito.

La prima regola d'oro è quella di non tentare nulla di nuovo nel bel mezzo di un programma lungo. In teoria il vostro programma sarà composto da subroutine su nastro che avete controllato da cima a fondo prima di unirle in un unico lungo programma. In realtà, le cose non sono così semplici, particolarmente quando le subroutine sono presenti solo sotto forma di linee di dati per i programmi in BASIC in cui compare l'istruzione POKE. Chi utilizza un assembler trarrà il meglio da esso, poiché sarà in grado di memorizzare le istruzioni del linguaggio Assembler come programmi BASIC ed unirle o correggerle a volontà.

La cosa migliore da farsi, oltre che tenere una libreria di subroutine su nastro, è di avere delle esaurienti note sulle subroutine. In aggiunta alle vostre, potete crearvi una libreria di routine rilevate da riviste specializzate. Anche se non le utilizzerete, il modo in cui vengono documentate dovrebbe fornirvi qualche idea di come tenere la documentazione delle vostre. Se avete intenzione di usare una nuova routine in un programma, sarà meglio provarla prima da sola, in modo tale da essere sicuri di quello che deve essere messo in ogni registro; questo, prima ancora che venga richiamata la subroutine e di ciò che sarà presente dopo nei registri.

Date uno sguardo ad alcuni esempi presi da una rivista e guardate come vengono presentate le informazioni. Una programmazione di questo tipo dovrebbe eliminare molti errori, ma se siete ancora alle prese con un programma che non funziona, e che non volete mettere in disparte, allora dovrete fare uso di un punto di arresto.

Un punto di arresto, per quanto riguarda il sistema operativo del C64,

è rappresentato dal byte $60. Questo è il byte di RTS, e il suo effetto è di riportare il programma in ambiente BASIC. Quando si è di nuovo in ambiente BASIC, si possono esaminare i contenuti della memoria tramite l'istruzione PEEK. Il principio consiste nell'andare a vedere un punto del programma in cui viene messo qualcosa nella memoria. Se mettete il byte $60 dopo questo punto, quando il programma verrà eseguito tornerà al BASIC immediatamente dopo aver usato la memoria. Usando l'istruzione PEEK, potrete quindi controllare che quanto era stato caricato nella memoria sia quello che vi aspettavate. In caso contrario, dovreste sapere dove andare a cercare l'errore. Se a questo punto tutto va bene, allora sostituite il byte originale di appartenenza al $60, e mettete $60 nell'indirizzo successivo che segue un'istruzione di scrittura in memoria. Tuttavia, questo tipo di operazione è più facile da portare a termine con l'aiuto di un buon programma supervisore, del quale parleremo poi.

L'errore più difficile da trovare usando questo o altri metodi è quello di un ciclo difettoso. Un ciclo difettoso causa sempre il blocco del computer. Sebbene i tasti RUN/STOP e RESTORE vi aiutino a venirne fuori, in questo caso non lo faranno mai. Ad esempio, è possibile che un programma alteri uno degli indirizzi che controllano l'uso della tastiera in modo tale che, sebbene riacquistiate il controllo del sistema, non possiate fare uso dei tasti. La funzione dei tasti RUN/STOP e RESTORE, ad esempio, può essere disattivata lavorando sulla locazione 808 (dec.). La ragione principale di questo genere di cose è un ciclo che salta indietro in un punto errato. Ad esempio, se abbiamo un programma di cui una parte è:

```
                LDX, #$FF
LOOP: (ciclo)   DEX
                BNE LOOP
```

potremo incontrare dei problemi. Supponiamo che esso sia stato assemblato a mano e che abbiamo fatto fare un salto indietro all'istruzione LDX invece che all'istruzione DEX. Questo errore fa sì che il valore del registro X rimanga immutato e non venga decrementato a 0. Un errore del genere viene facilmente localizzato in linguaggio assembly, poiché la posizione del nome dell'etichetta è facile da controllare. È molto più difficile da trovare quando il programma viene scritto in linguaggio macchina. Come sempre consigliamo di fare molta attenzione nei cicli e di seguire il metodo mostrato in questo libro, cioè calcolare e controllare gli scostamenti.

Il programma superiore TIM

Ho accennato brevemente in questo capitolo ai programmi monitor. Essi non hanno niente a che vedere col monitor della TV, che è un tipo di visore per i segnali di qualità superiore. Nell'ambito del software, "monitor" significa programma supervisore, cioè un programma che controlla ogni operazione di un programma in codice macchina. Un programma superiore è, o dovrebbe essere, un programma in linguaggio macchina che può essere caricato all'interno di una parte della memoria da voi non utilizzata che per questo scopo. Una volta caricato, permette di visualizzare il contenuto di qualsiasi parte della memoria (nella notazione esadecimale), di variare i contenuti di qualsiasi parte della RAM e di esaminare o variare il contenuto dei registri del microprocessore 6502. Queste sono le funzioni più elementari di un programma supervisore che si rivela utile se un programma può essere eseguito, se vi si possono inserire dei punti di arresto e se si possono esaminare i registri durante l'esecuzione.

Il programma monitor ideale è quello che esegue un passo alla volta un programma in codice macchina, visualizzando ad ogni passo i contenuti dei registri e della memoria. Un "monitor" di questo tipo era disponibile per i vecchi TRS-80 MK.1, e sarebbe un buon articolo per i seri programmatori in linguaggio macchina di altri computer.

Il monitor MIKRO

La cartuccia del monitor/assembler MIKRO contiene, fra le sue varie funzioni, un eccellente programma supervisore modellato sul programma monitor TIM disponibile sui più vecchi computer "PET". Il C64 deve essere spento nel momento in cui inserite la cartuccia ma, una volta inserita, può essere lasciata fino a quando non si abbia bisogno di usarne un'altra. Il semplice inserimento della cartuccia non causa l'esecuzione del programma. Esso deve essere richiamato battendo TIM e RETURN. In questo modo vedrete apparire sullo schermo ciò che è mostrato nella figura 8.1. In essa vengono mostrati l'indirizzo di potenza

```
CALL @    814B
  ADDR   IRQ     SR   AC   XR   YR   SP
.;814B   EA31    4D   00   00   00   FB
```

Figura 8.1 - Il messaggio del programma monitor TIM. Compare ogni volta che attivate il programma digitando TIM e premendo RETURN

di TIM e, sotto, il contenuto dei principali registri del 6502 accompagnati dagli indirizzi più importanti. Se per il momento tralasciamo gli indirizzi (a questo stadio dell'apprendimento dell'uso del linguaggio macchina non sono importanti), possiamo esaminare i registri. Esistono 5 contenuti principali dello STATUS REGISTER (SR), l'Accumulatore, l'Indice X, l'Indice Y e lo Stack Pointer (puntatore di pila). Se avete appena acceso il C64, l'accumulatore e i registri indice sono azzerati. Lo status register ha, di norma, il valore 4D (sono settati vari flag, incluso il flag carry), e il registro SP non è al suo valore di inizio $FF.

Per ora tutto questo non è di grande utilità, ma una volta che vi inoltrerete nella programmazione, troverete che queste informazioni vi sono di grande aiuto. Potrete richiamare la visualizzazione di questi dati ogni volta che lo desiderate, premendo la lettera "R" e RETURN. Tutte le operazioni del programma monitor possono essere richiamate usando la combinazione di un punto e di una lettera, come .R.

Mentre il programma monitor rimane in attesa di un altro comando, posiziona un punto sullo schermo così che dovete premere solo la lettera da voi prescelta, quindi il tasto RETURN. La figura 8.2 mostra le opzioni disponibili. Naturalmente non dovrete far uso di tutte queste opzioni e alcune di esse potrebbero perfino non riscuotere per voi grande

---

M - Mostra il contenuto della memoria in esadecimale. M deve essere seguito da un indirizzo di partenza e da uno di termine, usando 4 cifre esadecimali.

S - Salva un programma in codice macchina su nastro o su disco.

L - Carica un programma in codice macchina da nastro o da disco.

G - Esegue un programma. G deve essere seguito dall'indirizzo di partenza del programma, usando 4 cifre esadecimali.

H - Ricerca nella memoria una serie di byte. H deve essere seguito da un indirizzo di partenza e da uno di termine, e dai byte da trovare, tutti in esadecimale.

T - Trasferisce un blocco. T deve essere seguito dall'indirizzo di termine e dal nuovo indirizzo di partenza.

D - Converte il codice. Devono seguire l'indirizzo di partenza e quello di termine per la conversione. Lo schermo viene riempito e, per continuare la conversione, basta premere un tasto.

X - Ritorno al BASIC.

Figura 8.2 - Il menu TIM per il pacchetto MIKRO. Tipiche funzioni di un programma supervisore

interesse. Invece di esaminarle in ordine, sceglieremo nell'uso del codice macchina quelle più utili per il principiante. Fra questi, il comando "M" (esame della memoria) è il più importante. Quando premete .M (di solito solo M per le ragioni spiegate sopra), dovrete battere uno spazio. Deve quindi essere seguito dall'indirizzo di partenza della memoria che vi interessa e deve essere scritto sotto forma esadecimale di 4 cifre. Ad esempio, l'indirizzo 2B deve essere scritto 002B. Quindi battete un altro spazio, poi l'indirizzo di termine della parte di memoria che volete esaminare, che deve essere scritto sempre sotto forma di un numero esadecimale di quattro cifre.

Fino a quando non avrete premuto il tasto RETURN nulla accadrà, perciò fino a quel momento avrete la possibilità di cambiare idea. Premendo RETURN, vedrete apparire sullo schermo la sezione della memoria richiesta. La colonna a sinistra contiene un punto, un due punti, quindi l'indirizzo di partenza per ogni fila di numeri, ognuna delle quali contiene otto numeri. I numeri memorizzati in quegli indirizzi vengono mostrati in queste file. Supponiamo, ad esempio, di esaminare la fila il cui numero di partenza (a sinistra) sia 0040. Il primo byte è quello memorizzato all'indirizzo $0040; il byte successivo è quello memorizzato all'indirizzo $0041, e così via.

Quando TIM visualizza il contenuto della memoria in questo modo, si possono variare i contenuti della memoria stessa in qualsiasi indirizzo che appaia sullo schermo. Questa operazione può essere fatta alla stregua di una correzione in BASIC tramite l'edit. Si posiziona il cursore sulla prima cifra del numero da correggere tramite i tasti di controllo del cursore. Quindi si digita la cifra desiderata e si continua scrivendo la seconda. Naturalmente potete anche variare una sola cifra, che deve essere una cifra esadecimale valida. Dopo aver premuto il tasto RETURN, il valore sarà sostituito all'interno della memoria. Se battete una lettera non ammessa o che non ha senso, essa non verrà sostituita.

Una funzione utile, compresa nella possibilità di variare un codice, è che si può rendere qualsiasi valore uguale a $00. Questo rappresenta l'istruzione BRK (BREAK), che ha come effetto il ritorno al programma monitor TIM. Non è uguale a RTS, che riporta normalmente l'esecuzione del programma in ambiente BASIC.

Quando si usa il comando BRK per tornare al programma monitor, si può usare il comando .R per esaminare i registri del 6502 e il comando .M per vedere cosa è accaduto nella memoria. Lo scrivere il codice $00 in un programma in linguaggio macchina viene chiamato "disporre un punto d'arresto", ed è particolarmente utile quando si vuole sapere cosa sta facendo un programma a un particolare punto del suo percorso.

L'istruzione successiva più importante, per quanto ci riguarda, è .R,

già citata in precedenza. Premendo la lettera R (e RETURN) si visualizza il contenuto del microprocessore 6502. A meno che non sia stato interrotto il computer durante l'esecuzione di un programma, non c'è molto da vedere al suo interno se non quando utilizzerete le funzioni avanzate del programma monitor TIM e, come l'utilizzo del punto d'arresto, si rivelerà di grande aiuto. Vengono mostrati i contenuti di tutti i registri e, di norma, in particolare verranno presi in considerazione i valori all'interno dei registri A, B, X e Y. Ciò significa che, al momento dell'esecuzione del programma in codice macchina, con il punto di arresto inserito, il programma si bloccherà al punto di arresto e, quando usate l'istruzione .R, lo schermo mostrerà il contenuto dei registri. Questo molto spesso è quanto basta per capire dove il programma non ha funzionato bene. TIM vi permette di predisporre tanti punti di arresto quanti ne desiderate, sebbene possiate averne uno solo al posto di un codice operativo. Se, ad esempio, considerate una parte di programma in codice macchina per l'istruzione LDA $4050, questo è formato dal codice operativo $AD seguito dai byte $50 e $40. Se volete un break in questo punto, potete sostituire solo il codice operativo $AD con $00, e non gli indirizzi del byte. Solo i codici operativi rappresentano delle istruzioni per il microprocessore 6502.

Quando il programma si ferma a un punto di arresto, si può esaminare il contenuto dei registri, usare .M per esaminare la memoria, quindi far continuare l'esecuzione del programma premendo ".G". Si possono perfino modificare i contenuti dei registri prima di eseguire di nuovo il programma, sebbene per ora voi non abbiate l'esperienza sufficiente per compiere tali operazioni. È tuttavia un metodo comodo per controllare le operazioni di un ciclo, e per vedere cosa accade quando i valori di un registro raggiungono valori come $FF e $00. Invece di analizzare l'interno del ciclo dozzine di volte, potete farlo una sola volta per controllare che funzioni, quindi cambiare il contenuto dei registri in modo tale da fermarlo e quindi controllare che ciò avvenga. Il cambiamento del contenuto dei registri del 6502 avviene dopo aver letto i contenuti di tali registri tramite il comando .R. Posizionando il cursore sulla cifra da variare e battendo la nuova otterrete, dopo avere premuto il tasto RETURN, il cambiamento desiderato. Questo valore verrà immesso all'interno del registro quando si riprenderà l'esecuzione del programma.

".G" deve essere seguito da uno spazio, quindi da un numero a 4 cifre esadecimali relativo alla successiva istruzione che volete eseguire. Quando avete finito il controllo dovrete eliminare tutti i punti di arresto da voi immessi precedentemente usando l'istruzione .M e apportando le dovute variazioni.

Questi sono validi metodi di debugging e di ordinamento dei program-

mi, e pur tuttavia formano solo una parte delle funzioni che questo utilissimo programma monitor mette a disposizione. La mancanza delle funzioni SAVE e LOAD per i programmi in codice macchina è supplita, nel programma TIM, dai comandi .S e .L. Il comando .S è usato per salvare i programmi in linguaggio macchina, per il quale si deve specificare il nome del file se la memorizzazione avviene su nastro o su dischetto, l'indirizzo di inizio del programma e quello di termine +1. Tutti i numeri devono essere scritti in esadecimale, usando 4 cifre per gli indirizzi e due cifre per i singoli byte. Ad esempio, supponiamo di voler salvare su nastro un programma che inizia all'indirizzo $9F60 e finisce all'indirizzo $9FE2. Il comando SAVE verrebbe scritto:

.S"MCPROG",01,9F60,9FE3.

Di questo insieme, .S rappresenta il comando di salvataggio, e MCPROG è il nome del file. Il numero 01 indica che si sta usando la cassetta (08 significherebbe l'uso della periferica per i dischi). Dovete usare 01 e 08 e non 1 e 8. Poi segue l'indirizzo di partenza e quello di termine (aumentato di 1 unità). Le virgole vengono usate come separatori.

Caricare un programma da cassetta è molto più semplice. Tutto ciò che dovete scrivere è:

.L "MCPROG"

o solo .L se volete caricare il primo programma sul nastro. Per ultimo, ma non per questo di minore importanza, il programma monitor TIM permette la ricerca di byte, il loro trasferimento e la conversione della memoria. Supponiamo che vogliate verificare se il byte 2D si trova tra gli indirizzi 9F00 e 9FFF. Se battete .H 9F00 9FFF2D e il tasto RETURN, sullo schermo appariranno tutti gli indirizzi delle locazioni in cui compare questo byte. Si possono cercare anche gruppi di byte, il che si rivela spesso più utile. Ad esempio, se state cercando di caricare il valore dell'indirizzo 7A (codice 85 7A) nella ROM, e pensate che si trovi tra l'indirizzo $A000 e $AFFF, allora .H A000 AFFF 85 7A troverà i 10 indirizzi in cui compaiono queste combinazioni di codici. Se si vuole trasportare una parte della memoria RAM nel proprio programma senza copiarla byte per byte, si riutilizza il comando .T, che deve essere seguito dall'indirizzo di partenza dei byte che si vogliono copiare, quindi dall'indirizzo di termine, e per finire dal nuovo indirizzo di partenza. Se, ad esempio, volete trasportare i codici compresi tra gli indirizzi $AB7B e $ABA4 nella memoria RAM, o partire da $9F00, allora l'istruzione necessaria sarà:

.T AB7B ABA4 9F00

Per finire, il comando .D permette di disassemblare la memoria. È par-

ticolarmente utile per la ROM, se sullo schermo non appare alcuna conversione. Il comando .D deve essere seguito da un indirizzo di inizio e da uno di fine, entrambi sotto forma di numeri di quattro cifre esadecimali. Premendo RETURN ottterrete la visualizzazione della conversione su 25 linee (a meno che non si tratti di una conversione di pochi codici). Ogni linea contiene un numero di riferimento, un indirizzo (del byte del codice operativo), il codice esadecimale e la sua versione in linguaggio assembler. Alcune parti della ROM contengono solamente byte di dati, senza codici operativi, e se questi non sono numeri che possono essere interpretati come codici operativi, essi compaiono come istruzioni BYT. Se il disassemblaggio si estende a più di 25 linee, premendo qualsiasi tasto verranno visualizzate le successive 25 linee, fino a quando tutta la memoria richiesta non sia stata disassemblata. L'uso del disassembler è particolarmente utile se si vuole utilizzare qualche routine della ROM.

Se avete terminato di utilizzare il programma TIM, per tornare in ambiente BASIC, premete la lettera «X». Tutto sommato, è uno dei programmi più soddisfacenti e degni di nota poiché è solo una parte di un pacchetto che caratterizza il MIKRO Assembler oggetto del discorso che porteremo ora avanti.

Come usare il MIKRO Assembler

Durante la stesura di questo libro erano presenti diversi programmi assembler per il C64, ma il MIKRO Assembler era di gran lunga il prodotto da me più stimato. Sebbene questo sia un libro introduttivo per lettori che probabilmente non andranno mai oltre l'uso di un assemblatore, si rende necessaria una descrizione.

Se si evitasse di descrivere questo assembler, sarebbe come mettere i nomi di Alcock e Brown nella stesura della storia dell'aviazione, sebbene non molti lettori possano dire di avere avuto esperienza diretta delle loro imprese.

Qualsiasi assemblatore degno di questo nome viene trascritto anche in codice macchina. Il MIKRO Assembler fa di più poiché, essendo caricato su cartuccia, rappresenta un grande vantaggio. Un assembler che dovesse essere caricato da cassetta rappresenterebbe, invece, una seccatura. La ragione di questo è che inevitabilmente quando si sta svolgendo un programma in linguaggio macchina mediante un assemblatore, il programma scomparirà in qualche fase del controllo. Quando ciò accade, esso riesce a cambiare i valori memorizzati nella memoria, per cui sarà facile che alteri anche l'assembler, come qualsiasi altra cosa, se l'assembler è presente nella RAM.

Con l'assembler su cartuccia ROM, i suoi codici sono al sicuro e voi potrete quindi tornare ad esso in qualsiasi istante.

Ogni assembler si differenzia dagli altri, perciò è probabile che la mia descrizione sul funzionamento del MIKRO non sia esattamente uguale al modo di operare di altri assembler. I concetti basilari, tuttavia, sono gli stessi; è solo questione di imparare una diversa sequenza dei comandi. Le differenze tra assembler sono piuttosto simili alle differenze che intercorrono tra diversi computer, ma una volta che se ne conosce il linguaggio, assumono un'importanza secondaria. Il linguaggio, in questo caso, è l'assembler del 6502. Ciò che dovrete imparare è come scrivere un programma nella forma che il MIKRO può comprendere e trattare, trasformandolo in linguaggio macchina, e memorizzare il codice in modo tale che possa eseguire il programma.

Il funzionamento del MIKRO

Prima di acquisire una buona conoscenza del MIKRO, dovete sapere come affronta l'assemblaggio delle istruzioni. Il principio si basa sul concetto per cui si batte il programma in linguaggio assembler su linee numerate, proprio come scrivereste un programma in BASIC. La cosa non è casuale, perché il programma può essere scritto anche se non viene installata la cartuccia del MIKRO. State semplicemente usando i dispositivi del C64 per la scrittura di linee di programma. Purché non utilizziate il Commodore 64 per eseguirle, non avrete bisogno della presenza del MIKRO fino a quando non assemblerete il programma.

Le linee di programma in linguaggio macchina possono essere memorizzate come si salva un programma in BASIC. Ciò che distingue questo programma dal BASIC è che utilizza il linguaggio assembler, e contiene controlli per il programma assembler. Tuttavia è un vantaggio avere inserita la cartuccia dell'assembler MIKRO quando si batte il programma. Una buona ragione è che si può far uso dell'istruzione AUTO che permette la numerazione automatica delle linee di programma. Quando si fa uso di questo comando, sullo schermo appare la linea 100. È la linea di partenza del programma e, ogni volta che si termina di battere una linea, dopo avere premuto il tasto RETURN, apparirà la linea successiva. Le linee vengono incrementate di dieci in dieci. Questa funzione è talmente utile che quando scrivo un programma BASIC tengo sempre inserita nel computer la cartuccia del MIKRO.

Il comando AUTO può essere modificato per ottenere diversi numeri di linea di partenza e di incremento. Se, ad esempio, battete AUTO 10,5 otterrete un incremento di linea di 5 partendo dalla linea 10. Premendo il tasto RETURN uscirete dalla numerazione automatica.

Un altro comando veramente utile presente nel MIKRO è DELETE, che permette di cancellare il numero di linee voluto. Infatti se digitate: DELETE 100-120, le linee 100, 110 e 120 verranno cancellate dal programma. Digitando DELETE —120 cancellerete le linee con l'etichetta inferiore alla 120 inclusa, mentre DELETE 120— permetterà di cancellare le linee superiori alla 120 inclusa.

In aggiunta a questi comandi di editing, entrambi utilizzati nella stesura di programmi BASIC, il MIKRO, ne possiede altri due adatti particolarmente per la programmazione in linguaggio assembler.

FORMAT permette di sistemare ogni programma su delle colonne ordinate illustranti gli indirizzi, i contrassegni, i codici operativi, gli operandi e i commenti a parte, non importa l'ordine in cui li avete battuti originariamente. È particolarmente utile nella stesura di programmi in linguaggio assembler, difficilmente controllabili qualora non vengano scritti in modo chiaro e ordinato.

L'altro comando utile è FIND, che deve essere seguito da un nome, ad esempio un nome contrassegno, e che vi mostrerà tutte le linee in cui questo nome compare. FIND può essere utilizzato anche per i programmi BASIC, e si rivela molto utile per sapere dove avete usato dei nomi di variabili. Ad esempio, FIND X vi elencherà tutte le linee in cui compare la variabile X. La linea non viene visualizzata nel modo usato dal BASIC ma come se fosse un programma in linguaggio assembler. Si può limitare la ricerca all'interno di una serie di linee aggiungendo al comando FIND una virgola e i numeri di linea delimitatori separati da un trattino. Ad esempio, FIND LOOP, 150-300 elencherà ogni linea compresa tra la 150 e la 300 in cui compaia la variabile LOOP. Potete inoltre usare le stesse valide opportunità per DELETE e LIST, cioè le scritture FIND LOOP, 400— e FIND LOOP, —2000 vengono ammesse.

Altri due comandi, altrettanto utili, sono NUMBER e DISASSEMBLE. Il primo permette di visualizzare un numero nelle notazioni esadecimale, binaria e ottale (poco usata al giorno d'oggi). I numeri che vorrete visualizzare saranno contraddistinti, a seconda della notazione usata, da tre caratteri: $ per i numeri esadecimali, @ per quelli ottali, % per quelli binari. Se, ad esempio, digitate NUMBER123, otterrete:

```
          ESA (HEX)  =  $007B
DECIMALE (DECIMAL)  =   123
     OTTALE (OCTAL)  =  @000105
   BINARIO (BINARY)  =  %0000000001000101
```

Naturalmente potrete immettere il numero di cui volete la conversione in qualsiasi notazione. Perciò, scrivendo NUMBER$45 o NUMBER@215312 o NUMBER%1101110111 otterrete la visualizzazio-

ne di tutte le conversioni del numero immesso. Il comando DISASSEMBLE permette di ottenere la stessa operazione del comando .D del programma monitor TIM. Ora, il linguaggio assembler usato dal MIKRO si avvicina allo standard che la Mostek, i produttori del 6502, hanno progettato. Sebbene non sia perfettamente identico, le differenze sono minime. Per esempio, una di queste è data dai commenti che vengono separati da un punto esclamativo invece che da un punto e virgola. Daremo uno sguardo ad alcune differenze, poiché le altre sarebbero interessanti solo dopo avere maturato una certa esperienza sul linguaggio macchina.

L'altro punto in cui questo programma differisce da quello in BASIC sta nelle istruzioni all'assembler. Non basta, cioè, avere un set di istruzioni assembler poiché è necessario, ad esempio, indicare in quale indirizzo della memoria volete assemblare il codici dell'istruzione. Come spesso accade, però, scoprirete che un set standardizzato di istruzioni di questo tipo basterà praticamente per tutti i vostri scopi. L'assembler MIKRO non metterà dei codici agli indirizzi che avete usato, ad esempio, fino ad ora. E questo perché esso utilizza tali indirizzi per i suoi propri scopi.

4K di memoria sono lasciati a disposizione per le vostre routine a partire dall'indirizzo $C00. Oppure potete assemblare il codice all'indirizzo specificato usando POKE56,127. Questo è l'indirizzo $7F00, e per cominciare l'assemblamento del codice all'indirizzo successivo $7F01, dovrete iniziare il vostro programma in linguaggio assembler con:

```
100 *=$7F01
```

Normalmente si parte con l'allocare la memoria. Dovrete essere ancora sicuri di avere riservato questa memoria. Ciò può essere fatto sia battendo POKE56,127 in BASIC, oppure coi corrispondenti LDA #127 e STA 56 all'inizio del programma. Senza la presenza del BASIC, tutta la memoria RAM è a vostra disposizione. In tal caso è logico posizionare l'inizio del vostro programma all'indirizzo di inizio del BASIC. In questo modo potrete fare uso dei comandi LOAD e SAVE, se lo vorrete. Se non usate il comando di partenza *=$7F01 o comandi simili, il vostro programma verrà sempre assemblato a partire dall'indirizzo $033C. Questa è la parte della memoria RAM chiamata «buffer di cassetta». Viene usata solo nelle operazioni di caricamento e salvataggio, e si estende fino all'indirizzo $03FC. Tuttavia, non è proprio il luogo ideale per il vostro programma, se avete intenzione di usare i comandi LOAD e SAVE.

Dopo avere allocato l'indirizzo di partenza, potrete allocare le etichette (o contrassegni), le quali possono essere relative ad indirizzi o a byte. Ad esempio, potreste avere:

```
110 SCRNST  = $0400
120 CARRET  = $0D
```

come linee seguenti. Esse non genereranno alcun codice quando starete assemblando. Ciò che faranno è di assicurare che, in qualsiasi modo queste etichette compaiano, i numeri verranno ordinati. In ogni punto in cui apparirà SCRNST (inizio dello schermo), verrà messo il numero $0400, e ancora, in ogni punto in cui apparirà la parola CARRET (ritorno di carrello, RETURN), verrà messo il codice $0D. Usando i contrassegni in questo modo, si rende il programma più facile da seguire, mentre il definire all'inizio del programma queste parole assicura la vostra conoscenza della loro funzione senza dovere, per questo, cercare lungo l'intero programma.

Questi nomi contrassegni dovrebbero essere formati da sei lettere al massimo. Se dimenticherete di definire un qualsiasi nome, l'assemblaggio non potrà procedere, e voi otterrete un messaggio di errore quando cercherete di assemblare il programma. Un contrassegno non deve essere necessariamente definito in questo modo; può essere anche definito nel programma, ad esempio, da:

```
220 CICLO : LDA #124
```

Quindi possiamo scrivere un programma in linguaggio assembler come lo scriviamo in BASIC, ma per ogni linea di programma possiamo scrivere solo una istruzione. Ponendo al termine del programma l'istruzione RTS, si è sicuri che torni in ambiente BASIC, e che le linee del programma in assembler potranno essere seguite anche da linee di programma in BASIC. Si deve, tuttavia, porre una linea di termine alla fine della sezione in linguaggio assembler. Non è una cosa molto facile l'assemblare e il tornare in ambiente BASIC. Se non è presente il comando END, allora il MIKRO cercherà di leggere il BASIC e da ciò scaturirà un messaggio di errore. La differenza principale, infatti, sta nel fatto che il MIKRO è piuttosto meticoloso sul modo in cui viene scritto un programma in linguaggio assembler. Si deve, ad esempio, lasciare uno spazio tra ogni campo di linguaggio assembler. Per "campo" si intende una sezione che può essere rappresentata da un contrassegno, da un codice operativo o da un commento. Non è possibile, ad esempio, scrivere:

```
CICLOLDA #$45!INIZIO
```

e sperare che esso venga eseguito come vengono eseguiti i comandi dal BASIC, in cui le istruzioni possono essere scritte senza rispetutare regole di spaziatura tra parole riservate.

Normalmente otterrete il messaggio di errore: "NO OPCODE", che significa che l'assembler non è in grado di distinguere le istruzioni se non

vengono interspaziate. Se, invece, battete questi campi separandoli tramite uno spazio, cioè:

```
CICLO LDA #45 ! INIZIO
```

l'assembler sarà in grado di tradurli esattamente in linguaggio macchina.

Nel corso di questo libro daremo delle informazioni che permetteranno di aumentare il gruppo di istruzioni, sebbene ciò che abbiamo illustrato sinora rappresenti la base per la vostra conoscenza e per un migliore utilizzo del MIKRO. Se ciascun programma inizia con a*= per l'allocazione della memoria, quindi definisce il numero più grande di contrassegni (LABEL) possibile, significa che non avete fatto le cose frettolosamente. Se seguite questi preliminari, sarete in grado di battere le linee di programma in assembler. Se fate uso di un numero senza farlo precedere da alcun simbolo (come ad esempio $), esso verrà considerato come numero decimale. Usando il simbolo $ si parlerà di numeri esadecimali, con @ ottali e con % binari, come già visto precedentemente parlando del comando NUMBER. L'asterisco indica sempre "l'indirizzo attuale". Questo è l'indirizzo d'inizio dell'istruzione in cui appare l'asterisco, così che l'asterisco si rivela un modo utile per indicare un indirizzo senza dovere sapere in realtà quale esso sia. Il punto esclamativo permette l'aggiunta di commenti ad una linea di programma, alla stregua dell'istruzione REM del BASIC.

Un assembler permette anche l'uso di quelle che vengono chiamate "direttive assembler", o "pseudo-codici operativi", cioè frasi dirette al compilatore assembler. Il MIKRO non fa eccezioni, per cui vale la pena di vedere ora alcune di queste direttive poiché esse (o versioni di esse) vengono largamente utilizzate sugli assemblatori di altri microprocessori.

Abbiamo già visto il carattere *, che indica l'indirizzo attuale. Tre altri pseudo-codici operativi sono WOR, BYT e TXT, i quali vengono tutti usati per registrare nella memoria dei byte che non siano istruzioni. WOR registra una "parola" di due byte. Esso separerà il numero in byte alto e byte basso, per poi memorizzarli nell'ordine basso-alto (LO-HI). WOR $7F12, ad esempio, mette nella memoria $12 e $7F, in quest'ordine. Questi vengono memorizzati a partire dall'"indirizzo corrente". Supponiamo, ad esempio, di volere che un programma faccia uso di una tavola di valori che iniziano all'indirizzo $9FF0. Potreste includere nel vostro programma assemblatore le linee:

```
200*=$7FF1
210 WOR$2E14,$115B,$513A
```

e questo memorizzerebbe, portando dall'indirizzo $7FF1, la sequenza

14,2E,5B,11,3A,51. Da notare l'uso multiplo di indirizzi dopo la parola WOR, purché separati da una virgola.

Lo pseudo-codice BYT svolge la stessa funzione, ma per un solo byte, perciò BYT $0D posizionerà il codice del carrello di ritorno all'indirizzo corrente. BYT può scrivere anche il codice ASCII delle lettere tramite il carattere ' (apostrofo). Se, ad esempio, scrivete BYT'A nel programma, allora il codice ASCII della lettera A viene memorizzato nell'indirizzo attuale. TXT è il metodo più conveniente per memorizzare molte lettere. Dopo TXT potete mettere una stringa di lettere, usando le virgolette all'inizio e alla fine. I codici ASCII di ogni lettera, inclusi gli spazi e i segni di punteggiatura, vengono immessi nella memoria a partire dall'indirizzo attuale. Ad esempio, scrivendo TXT "PREMERE QUALSIASI TASTO" si memorizzeranno tutti i codici ASCII dei caratteri che compaiono all'interno delle virgolette.

Come farlo funzionare

Quando un programma assemblatore è stato battuto, può essere salvato su cassetta come un qualsiasi programma in linguaggio BASIC. Questo passo è importante, poiché rappresenta il vostro "programma sorgente".

Per "programma sorgente" si intende il set di istruzioni che assembleranno il programma in linguaggio macchina. Se ne avete una registrazione, lo potrete ricaricare, correggere e riassemblare a vostro piacere. Poiché il programma sorgente può essere registrato come un programma in BASIC, dovete prestare attenzione nel contrassegnare le cassette o i dischetti in modo da non confonderli. Il programma sorgente non verrà eseguito quando si digiterà il comando RUN. Per assemblare il vostro programma dovete battere ASSEMBLE, quindi premere il tasto RETURN. L'assembler passerà tre volte sul programma sorgente. Questa operazione si rende necessaria perché certi nomi contrassegno non possono essere definiti all'inizio del programma. Qualsiasi errore trovato sarà rivelato nel primo e nel secondo passo attraverso il programma sorgente, e l'assemblaggio avrà termine con un messaggio d'errore. Il messaggio di errore fornirà, come vi aspetterete, il tipo di errore e a quale linea esso compare. Se non sono presenti degli errori, apparirà scritto sullo schermo:

```
ASSEMBLE
* PASS(1)*
* PASS(2)*
```

```
  * PASS(3)*
***** ASSEMBLY COMPLETE *****
  START ADDRESS - $9F01
  END ADDRESS   - $9F43
```

Gli indirizzi sono molto utili se farete uso del comando SAVE del programma monitor TIM per registrare il programma in codice macchina su nastro o disco. L'indirizzo di partenza è anche quello di cui avrete bisogno quando richiamerete il programma in codice macchina in azione usando SYS. È da notare che l'assembler MIKRO non fa eseguire il programma sorgente, a meno che non compaia SYS seguito dall'esatto indirizzo di partenza, nell'ultima linea del programma sorgente.

Questo capitolo, tuttavia, non intende descrivervi completamente l'assembler MIKRO. È mia intenzione darvi un saggio di ciò a cui può assomigliare l'utilizzazione di un assembler. Se e quando sarete pronti all'uso del MIKRO, sarete in grado di dare un senso al manuale con le brevi istruzioni che lo accompagnano. Ancora più importante: sarete in grado di usare altri assembler, perfino assembler per altri microprocessori, se mai cambierete il vostro computer.

Capitolo 9

Un ultimo discorso

Uno dei problemi principali che nascono scrivendo un libro che riguarda il codice macchina per principianti è sapere dove fermarsi. Potrebbero essere scritti volumi per la programmazione del codice macchina del C64, e lasciare ancora spazio in modo che ogni argomento possa venir concluso in modo arbitrario. Il mio intento è stato quello di introdurre l'argomento e portarvi ad un livello dal quale possiate iniziare da soli a fare progressi. Una volta raggiunto questo stadio, potete fare uso di altri libri che sono disponibili e che trattano il codice macchina ad un livello più avanzato.

Il presente capitolo illustra alcune altre istruzioni e l'uso di alcune caratteristiche operative del C64.

Lo stack (pila)

Non si può procedere nel programmare un codice macchina senza passare attraverso la parola stack. Uno "stack" è una parte della memoria e il suo uso particolare è quello di preservare i byte che sono stati messi nei registri. Non c'è una serie speciale di chip che usiamo come stack, ma tratteremo isolatamente una parte della RAM a questo proposito. Per un microprocessore 6502 dobbiamo usare la memoria nella serie di indirizzo $100 e $1FF. Quello che probabilmente troveremo difficile da capire adesso è perché dovremmo avere bisogno di usare la memoria in questo modo.

Prendiamo un semplice esempio. Supponiamo di avere un programma nel quale si sta usando il registro A per tenere un codice ASCII per un carattere. Ora supponiamo, nel mezzo di questo programma, di voler produrre un ritardo di tempo facendo uso di un conto alla rovescia nel registro A. Ogni volta che portiamo un valore contatore in un registro A, sostituiremo il numero del codice ASCII che vi era stato messo

e, se proviamo a usare di nuovo il registro A nel resto del programma, dovremo ricaricarvi l'indirizzo. A questo serve lo stack. Per mezzo di una istruzione di un singolo byte, si possono memorizzare i contenuti dell'accumulatore o dello status register nello stack. Usando anche un'altra istruzione del genere, possiamo riportare di nuovi i valori nei corretti registri. L'operazione di portare il contenuto nel registro nello stack viene chiamata "PUSH", mentre l'operazione inversa viene chiamata "PULL".

Vedremo tra breve un programma di esempio che fa uso dello stack, ma per ora ho intenzione di ritornare ad argomenti più semplici. Il resto di questo capitolo, infatti, sarà rivolto ad esempi di programmi che serviranno come base per sviluppare routine veramente utili per il nostro C64. Devo precisare a questo punto che ora avete la base di lancio per quanto riguarda il codice macchina. D'ora in poi quello che occorre è la pratica e tante informazioni. Analizziamo attentamente ogni programma del C64 che contiene il codice macchina, per esempio. Anche se il codice macchina è nella forma di byte che sono messi nella memoria, possiamo smontarli e scoprire cosa fanno. Facendo questo, spesso possiamo scoprire indirizzi che ci saranno molto utili nei nostri programmi. D'ora in avanti, ogni cosa potenzialmente ci è utile!

Routine del tasto BEEP

Analizziamo ora una routine che illustra molti punti e che serve anche per introdurci ad una programmazione più avanzata. L'intenzione stavolta è di usare un programma che provocherà un breve beep ogni volta che premiamo un tasto. Questo accadrebbe quando stiamo lavorando normalmente nel BASIC così, quello che ci occorre, è un modo di inserire un pezzo extra di codice macchina nel BASIC. Questa è cosa diversa dal creare un programma di codice macchina che viene eseguito e che poi ritorna al BASIC, e dobbiamo sapere ben di più riguardo il C64 per essere capaci di usarlo.

Per iniziare, abbiamo il problema di "irrompere" nelle routine che il BASIC usa. Per fortuna il C64 usa una "scatola di congiunzione" particolarmente valida. Quello che voglio indicare con scatola di congiunzione è una parte di codice posta nella RAM invece che nella ROM. Ogni codice messo nella RAM può essere variato diversamente che nella ROM. Lo scopo di tutto questo è di poter avere delle correzioni fuori sequenza, cioè l'inserimento di parte di un programma nella routine usata dal sistema operativo del C64.

La ricerca di un luogo in cui effettuare questa operazione è la cosa

più difficile a farsi. È presente una routine, allocata all'indirizzo $0324 e $0325 che può essere utilizzata in questo modo, sebbene venga eseguita solo dopo avere premuto il tasto RETURN. Essa non è molto adatta al nostro caso, perché per il programma che ci siamo preposti vogliamo entrare in una parte del programma che è eseguita ogni volta che si preme un tasto. È proprio impossibile, lavorando da soli, scoprire indirizzi soddisfacenti in un tempo ragionevole: ci occorre così una lista smontata a tale scopo. Uno sguardo attento alla lista rivela un interessante paio di indirizzi a $028F e $0290. Questi due contengono un altro indirizzo, $E B48 che è l'indirizzo della pratica di decodifica della tastiera. Quando un tasto è premuto sulla tastiera causa segnali elettrici per arrivare a una porta. Il 6502 deve leggerli e convertirli in un codice singolo-byte usando un codice diverso per ogni tasto. Ovviamente il C64 fa questa azione separatamente: prima controlla se un tasto è premuto e poi lo decodifica. Uno sguardo a un disassembly della pratica della tastiera (che inizia a $EA87) mostra che questo indirizzo di $028F è usato indirettamente. Ciò significa che il C64 legge i byte in $028F e $0290 ogni volta che è premuto un tasto, e salta all'indirizzo dato da questi due byte.

Questo è ciò che stiamo cercando, ed è il luogo dove possiamo fare il nostro lavoro.

Quello che faremo è questo. All'indirizzo $028F sostituiremo l'originale byte di $48 con il basso byte di un nuovo indirizzo. All'indirizzo $0290 metteremo il byte alto di questo stesso nuovo indirizzo. Potete scrivere una routine che inizi da questo nuovo indirizzo. Alla fine, avremo un'istruzione JMP. L'indirizzo che segue JMP sarà la routine corretta di decodifica della tastiera $EB48. In questo modo ogni volta che si preme un tasto, la nostra routine viene eseguita prima che la macchina inizi a lavorare sulla decodificazione del tasto. La routine che abbiamo intenzione di sistemare è molto semplice. Spingerò i valori corretti per un suono negli indirizzi del suono chip, proprio come dovrebbe fare un programma BASIC. Dovrò includere una pratica di ritardo nel codice macchina così che il suono duri abbastanza da essere sentito, e finisca ritornando all'indirizzo $EB48. Tutto appare facile (quando lo si legge), ma sto per introdurvi molti nuovi concetti. Alcuni di questi sono nuove istruzioni per codificare, altri sono il risultato di nuove informazioni riguardo il C64. A questo punto ogni cosa è utile!

Inizieremo con un diagramma di flusso. La figura 9.1 ne mostra uno e, come ogni buon diagramma, non specifica particolarmente quello che abbiamo intenzione di fare. Immediatamente dopo l'inizio è usato il passo decisivo "è premuto, un tasto?". Questo ora si è occupato del sistema operativo, perciò non dobbiamo preoccuparcene ulteriormente. Se un tasto è premuto carichiamo gli indirizzi di suono. Ciò è tedioso più che diffi-

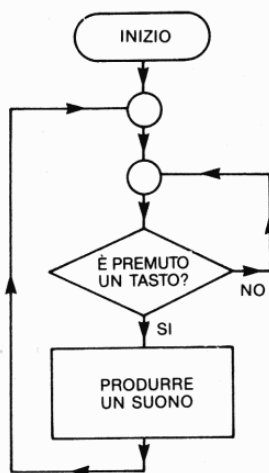



Figura 9.1 - Il diagramma per un programma 'beep' che suona una nota quando viene premuto un tasto

coltoso; è qualcosa che possiamo tenere su nastro pronto per l'uso, dopo averlo registrato.

Questo carico è seguito da un ritardo, poi l'azione del "suono pulito" spinge gli zero in alcuni degli indirizzi del suono. Se non lo facciamo, allora il suono continuerà dopo che il tasto è stato premuto e questo non è ciò che vogliamo. Alla fine torniamo al BASIC pronti per il tasto successivo.

La figura 9.2 mostra la versione del linguaggio assembly che è stato scritto come una serie di numeri di linea nella forma che usa il montatore MIKRO. Possiamo assemblarlo noi se desideriamo far pratica, ma la figura 9.3 mostra chiaramente il printout da un montaggio MIKRO e gli indirizzi e i byte di codice. Ricordiamo che l'indirizzo di inizio deve essere $7F00 quando il caricatore MIKRO è installato. Potete, se volete, scriverlo per adattarlo ai numeri di indirizzo maggiori se non state usando MIKRO. Se volete usare un altro indirizzo di inizio, una parte del codice deve essere alterata. La parte critica è l'indirizzo $7F3C per le subroutine di ritardo. Se cambiamo l'indirizzo al quale questo programma inizia, allora anche l'indirizzo delle subroutine di ritardo deve essere cambiato.

Dettagli, dettagli

Analizzando dettagliatamente il linguaggio del montaggio iniziamo

```
100 *=$7F00
120 LDA #<POINT
130 STA $28F
140 LDA #>POINT
150 STA $290
160 RTS
170 POINT PHA
175 LDA #15
180 STA 54296
190 LDA #190
200 STA 54273
205 LDA #248
210 STA 54278
220 LDA #17
230 STA 54273
240 LDA #37
250 STA 54272
260 LDA #17
270 STA 54276
280 JSR DELAY
290 LDA #0
300 STA 54276
310 STA 54277
320 STA 54278
330 PLA
340 JMP $EB48
350 DELAY LDA #100
360 STA 251
370 LOOP1 STA 252
380 LOOP2 DEC 252
390 BNE LOOP2
400 DEC 251
410 BNE LOOP1
420 RTS
430 END
```

Figura 9.2 - Il linguaggio assembly per una routine 'beep'. È stata scritta nella forma di linee numerate per l'assembler MIKRO

nel solito modo mettendo l'indirizzo di $7F00. La prossima parte del programma mette un nuovo indirizzo in $028F e $0290. Questo nuovo indirizzo sarà quello d'inizio della routine del suono. Mentre scriviamo queste prime linee, non sappiamo cosa sarà questo indirizzo perciò usiamo la definizione POINT. L'uso dei simboli < e > non è limitato a MIK-

```
100 7F00              *=$7F00
120 7F00 A90B                    LDA #<POINT
130 7F02 8D8F02                  STA $28F
140 7F05 A97F                    LDA #>POINT
150 7F07 8D9002                  STA $290
160 7F0A 60                      RTS
170 7F0B 48         POINT        PHA
175 7F0C A90F                    LDA #15
180 7F0E 8D18D4                  STA 54296
190 7F11 A9BE                    LDA #190
200 7F13 8D01D4                  STA 54273
205 7F16 A9F8                    LDA #248
210 7F18 8D06D4                  STA 54278
220 7F1B A911                    LDA #17
230 7F1D 8D01D4                  STA 54273
240 7F20 A925                    LDA #37
250 7F22 8D00D4                  STA 54272
260 7F25 A911                    LDA #17
270 7F27 8D04D4                  STA 54276
280 7F2A 203C7F                  JSR DELAY
290 7F2D A900                    LDA #0
300 7F2F 8D04D4                  STA 54276
310 7F32 8D05D4                  STA 54277
320 7F35 8D06D4                  STA 54278
330 7F38 68                      PLA
340 7F39 4C48EB                  JMP $EB48
350 7F3C A964       DELAY        LDA #100
360 7F3E 85FB                    STA 251
370 7F40 85FC       LOOP1        STA 252
380 7F42 C6FC       LOOP2        DEC 252
390 7F44 D0FC                    BNE LOOP2
400 7F46 C6FB                    DEC 251
410 7F48 D0F6                    BNE LOOP1
420 7F4A 60                      RTS
```

Figura 9.3 - Emissione dell'assembler MIKRO, che illustra i codici esa come il linguaggio assembly

RO; troveremo questi simboli in molti programmi 6502 nel linguaggio del montaggio. Il segno < significa "byte minore di" e il segno > significa "byte maggiore di" perciò quello che faremo nelle linee 120 e 150 è caricare i due byte dell'indirizzo di POINT negli indirizzi "function

box" di $028F e $0290. Dopo di che, ogni pressione di un tasto farà andare la macchina all'indirizzo di POINT.

Ora vogliamo eseguire questa parte "fissata" solamente una volta. Fissati questi indirizzi, vogliamo ritornare al BASIC eseguendo il resto del programma solamente quando un tasto è stato premuto. La prossima istruzione nella linea 160 è RTS che ritorna al controllo di BASIC. Il resto del programma è il bit che è usato ogni volta che un tasto è stato premuto.

La parte beep del programma inizia alla linea 170 con la parola definizione POINT. Questo garantisce che quando assembliamo, l'indirizzo della prima istruzione sarà posta nel "function box". La prima istruzione è PHA che significa "spingere l'accumulatore sulla pila". Quando la macchina vuole che il risultato della battuta di un tasto sia decodificato, avrà il risultato nell'accumulatore. Se distruggiamo quel byte, allora non ci possiamo aspettare che la macchina funzioni normalmente. Usando PHA preserviamo il contenuto dell'accumulatore sullo stack. Possiamo poi riacquistarlo alla fine della nostra pratica in modo che esso sia pronto nell'accumulatore quando saltiamo alla pratica decodificante, proprio come se il nostro programma non fosse esistito. Le linee da 175 a 270, poi, spingono i byte negli indirizzi del suono per produrre un beep basso-regolare. Ovviamente possiamo fare esperimenti con questi valori. Gli indirizzi sono gli stessi mostrati nel manuale C64 BASIC.

Il passo successivo è un ritardo. È stata usata una subroutine sebbene il codice avrebbe potuto essere messo nella parte principale del programma. La subroutine è una pratica familiare di ritardo che usa gli indirizzi di pagina 0. Ho evitato di usare le registrazioni X o Y nel caso esse fossero già state usate per la routine di decodificazione tasto. La ragione per cui si consiglia di stare attenti è che non ci sono istruzioni per salvare le registrazioni X o Y sullo stack. Il solo modo in cui questo può essere fatto è trasferire i byte all'accumulatore e poi "spingere" l'accumulatore. Quando i byte sono tolti dallo stack ritornano all'accumulatore (se è usato PLA) e possono poi essere trasferiti ad altri registri. Questo è un lavoro talmente tedioso che è preferibile cercare di evitare di doverlo fare. Io l'ho fatto usando gli indirizzi di pagina 0 per memorizzare i numeri. Un valore di 100 (decimale) in memoria produce un beep ragionevolmente breve.

Infine, il programma termina mettendo 0 in tre indirizzi del suono per fermarlo. Il valore che era originalmente nell'accumulatore ora è reintegrato dall'istruzione PLA e la routine termina con un salto all'indirizzo $EB48 per decodificare il tasto.

Quando questo programma è riunito e viene usato SYS32512 per iniziarlo, si otterrà un beep (e un ritardo) premendo qualsiasi tasto. Se ciò

non accade, forse avete dimenticato che il suono viene dall'altoparlante della TV e perciò il volume della TV deve essere regolato bene. Questa routine ci potrebbe avviare verso diverse piste nuove. Sareste in grado di fare un programma per assistere un utente cieco, dando una nota diversa per ogni tasto premuto? Significherebbe usare il codice che è nel registro A nel momento in cui i registri sono spinti per modificare il byte "pitch" (tono) che mettiamo nella registrazione B.

Rinumerando — un esempio di progetto di programma

Come grande finale illustrerò un progetto di un programma più elaborato dall'inizio alla fine. In effetti nessun programma è veramente finito finché possiamo giurare che non è più possibile migliorarlo. Spero infatti che troverete molto da migliorare in modo che questo programma possa servire da guida per cose migliori, anziché essere qualcosa che vogliamo a tutti i costi montare ed usare. Il programma è una cosa utile di cui il C64 è carente. L'idea è che possiamo mettere in memoria un numero di inizio e un numero di incremento e avere le linee di un programma BASIC rinumerate per noi, usando un indirizzo SYS.

Come al solito è meglio iniziare con qualche progetto. L'idea base è che possiamo sempre ottenere l'indirizzo del primo byte di un programma BASIC dagli indirizzi $2B e $2C. La linea successiva inizia ad un indirizzo fornito dai primi due byte di questa linea e il numero di linea è fornito dai successivi due byte. Dovremmo tuttavia essere capaci di programmare un programma ciclico che va da una linea alla successiva, cambiando i numeri, finché non trova che l'indirizzo seguente è 0000. Quel-

1. Mettere l'indirizzo del punto byte del programma BASIC.
2. Salvare il primo byte LB. Questo è il byte piccolo dell'indirizzo dell'inizio della riga seguente.
3. Salvare il secondo byte HB. Questo è il byte più alto dell'indirizzo di inizio della linea seguente.
4. Mettere il byte più basso di numero di linea nel terzo indirizzo della linea.
5. Mettere il byte più alto di numero di linea nel quarto indirizzo della linea.
6. Aggiungere il numero di incremento di linea al numero fornito di linea di inizio.
7. Esaminare l'indirizzo della linea seguente (fornita). Se questo è 0000, allora fermare.
8. Ripetere, usando l'indirizzo della nuova linea presa da HB e LB.

Figura 9.4 - Un sommario di quello che il programma di semplice rinumerazione dovrebbe fare

la è la fine del programma, i suoni sono abbastanza ragionevoli e la fig. 9.4 lo riassume.

Il prossimo passo, come dovremmo sapere, è disegnare un diagramma di flusso come mostrato nella figura 9.5. Il diagramma mostra più di quello che vogliamo fare, e questa volta vi ho aggiunto qualcosa in più. Occorre tenere "l'indirizzo della linea presente" in memoria, preferibilmente nei due numeri di pagina zero. Ho classificato queste AD e AD+1. Dobbiamo tenere anche il "numero dell'attuale nuova linea" in una coppia di indirizzi e questi sono classificati LIN e LIN+1. Alla fine occorre un incremento. C'è solamente un byte lasciato alla 255, perciò specificheremo che non possiamo rinumerare in incrementi maggiori di 255.

Figura 9.5 - Un diagramma di flusso per un programma di rinumerazione

Ora voglio precisare che questo è un programma semplice. Esso rinumererà solamente le linee stesse e non modificherà i numeri GOTO o GOSUB. Un esercizio di linguaggio di progetto macchina è vantaggioso ed utile. Scrivere per l'assembler MIKRO è l'ideale per rinumerare il tipo di programmi necessari.

La grande rinumerazione

Il programma è mostrato nella forma MIKRO nella fig. 9.6 ed è considerevolmente più lungo e più elaborato di qualsiasi cosa abbiamo fatto sinora. Tuttavia, voglio andare avanti spiegando dettagliatamente e indicando il perché di ogni passo. Molto raramente troverete una spiegazione dettagliata di come lavora un programma di codice macchina, così fate attenzione. Vi troverete presto a cercare di seguire dei programmi sulle riviste e queste spiegazioni saranno di vantaggio per tale lavoro.

Il programma inizia nel solito modo definendo un indirizzo di partenza e dei valori per alcune definizioni. L'indirizzo di inizio è stato scelto come $7F00 perché ho usato MIKRO per sviluppare questo programma. Qualsiasi valore di inizio usiamo, dobbiamo essere sicuri che la memoria sia protetta. Quando il programma è caricato in memoria protetta, caricare un programma BASIC non disturberà i byte del codice macchina. Se dimentichiamo di fissare la cima della memoria con un POKE all'indirizzo 56, nasceranno dei problemi.

La definizione AD è l'indirizzo di pagina zero usato per il piccolo byte dell'indirizzo della linea sulla quale il programma sta correttamente lavorando. Questo indirizzo cambierà da linea a linea, perciò faremo molto uso di AD.

INC è un byte di incremento, per cui i numeri di linea aumenteranno. L'incremento di solito sarà 10, ma il programma diventa più flessibile se possiamo cambiare tale quantità. Ogni cosa da 1 a 255 può essere usata. LIN tiene il byte basso del numero di linea per ogni linea rinumerata.

Proseguendo nel fissare questa parte, iniziamo il programma stesso. Il bit carry è azzerato e l'accumulatore è caricato dall'indirizzo 43. Questo è il byte basso dell'indirizzo del primo byte del BASIC e lo mettiamo nell'indirizzo della "linea attuale" AD. Poi carichiamo l'accumulatore da 44 che contiene il byte alto dall'inizio dell'indirizzo BASIC e lo mettiamo in AD+1. In questo modo AD ed AD+1 riportano una copia di 43 e 44, i byte dell'indirizzo del primo byte del nostro programma BASIC.

Ora inizia il ciclo e la registrazione Y è azzerata. LDA (AD) e Y caricheranno l'accumulatore dall'indirizzo tenuto in AD e AD+1. Questo è il primo byte del programma BASIC ed è il byte basso dell'indirizzo della linea successiva. Memorizziamolo sullo stack usando PHA e poi

```
100 *=$7F00
110 AD=251
120 INC=255
130 LIN=253
140 CLC
150 LDA 43
160 STA AD
170 LDA 44
180 STA AD+1
190 LOOP LDY #0
200 LDA (AD),Y
210 PHA
220 INY
230 LDA (AD),Y
240 PHA
250 INY
260 LDA LIN
270 STA (AD),Y
280 INY
290 LDA LIN+1
300 STA (AD),Y
310 LDA LIN
320 ADC INC
330 STA LIN
340 BCC NXT
350 INC LIN+1
360 CLC
370 NXT PLA
380 STA AD+1
390 PLA
400 STA AD
410 BNE LOOP
420 LDA AD+1
430 BNE LOOP
440 RTS
450 END
```

Figura 9.6 - Listato di linguaggio assembly per programma di rinumerazione. Questa è la forma usata dall'assembler MIKRO

incrementiamo la registrazione Y. Dato che la registrazione Y è stata incrementata, la prossima LDA (AD) Y caricherà dall'indirizzo del secondo byte nella linea BASIC. Questo è il byte alto dell'indirizzo della linea successiva ed è spinto sullo stack vicino al byte basso. Quindi viene di

nuovo incrementata la registrazione Y. Il prossimo passo è ottenere il primo byte basso del numero di linea da LIN. Questo è memorizzato nel successivo byte della linea, usando di nuovo il metodo indiretto di indirizzo Y. Se il registro Y è di nuovo incrementato, il byte alto del numero di linea di inizio è caricato nell'accumulatore e memorizzato nella linea usando gli indirizzi indiretti. Ciò completa la rinumerazione della prima linea con il primo dei numeri di linea che vogliamo.

Bisogna quindi aggiungere l'incremento al numero di linea. Carichiamo l'accumulatore da LIN e aggiungiamo l'incremento da INC. Il numero dell'accumulatore (il byte basso) è portato indietro in LIN e se c'è un riporto il byte alto a LIN+1 è incrementato. Il riporto viene eliminato per sicurezza. Ora dobbiamo mettere il prossimo indirizzo di numero di linea negli indirizzi AD e AD+1 per poter ripetere i processi. Il byte alto del successivo numero di linea è stato spinto per ultimo nello stack, così viene fuori per primo con l'istruzione PLA. È immagazzinato in AD+1.

Il byte basso viene tirato fuori e immagazzinato in AD. Possiamo esaminare a questo punto se il byte basso del successivo indirizzo è 0. Se non lo è, allora non abbiamo raggiunto ancora la fine del programma e dobbiamo tornare indietro. Se questo byte è zero dobbiamo esaminare il byte alto prendendolo da AD+1. Se questo byte non è zero non abbiamo ancora finito. Se entrambi i byte sono zero, abbiamo raggiunto la fine del programma e RTS ci riporta al BASIC.

Come dobbiamo usarlo? Serbiamo la memoria e mettiamo il codice macchina a posto. Carichiamo il nostro programma BASIC e decidiamo il numero di linea di inizio e un incremento. Inseriamo l'inizio di 253, e 254 con il byte alto in 254. Per un numero di inizio minore di 255, come 10 o 100, dobbiamo solo inserire un numero in 253, ma è più sicuro inserire 0 in 254. Se vogliamo iniziare con la linea 1000 dovremo esprimerlo come due byte. La fig. 9.7 mostra come fare. Poi mettiamo l'incremento (di solito 0) nell'indirizzo 255. Se abbiamo assemblato a $7F00 allora attiviamo il programma usando SYS325IL. In un batter d'occhio il nostro programma BASIC è rinumerato. Se abbiamo assem-

---

1000 come numero di linea iniziale consiste di due byte. Dividere 1000 per 256

dà come risultato 3.90625. Il numero intero è 3. Ora, 1000—3*256, che è 232.232,

è il byte basso del numero di linea.

Figura 9.7 - Dividere un numero di linea come 1000 in due byte

```
100 7F00            *=$7F00
110 7F00            AD          =  251
120 7F00            INC         =  255
130 7F00            LIN         =  253
140 7F00 18                     CLC
150 7F01 A52B                   LDA 43
160 7F03 85FB                   STA AD
170 7F05 A52C                   LDA 44
180 7F07 85FC                   STA AD+1
190 7F09 A000       LOOP        LDY #0
200 7F0B B1FB                   LDA (AD),Y
210 7F0D 48                     PHA
220 7F0E C8                     INY
230 7F0F B1FB                   LDA (AD),Y
240 7F11 48                     PHA
250 7F12 C8                     INY
260 7F13 A5FD                   LDA LIN
270 7F15 91FB                   STA (AD),Y
280 7F17 C8                     INY
290 7F18 A5FE                   LDA LIN+1
300 7F1A 91FB                   STA (AD),Y
310 7F1C A5FD                   LDA LIN
320 7F1E 65FF                   ADC INC
330 7F20 85FD                   STA LIN
340 7F22 9003                   BCC NXT
350 7F24 E6F6                   INC LIN+1
360 7F26 18                     CLC
370 7F27 68         NXT         PLA
380 7F28 85FC                   STA AD+1
390 7F2A 68                     PLA
400 7F2B 85FB                   STA AD
410 7F2D D0DA                   BNE LOOP
420 7F2F A5FC                   LDA AD+1
430 7F31 D0D6                   BNE LOOP
440 7F33 60                     RTS
```

Figura 9.8 - L'assemblatore MIKRO. Emissione per il programma di rinumerazione

blato a ogni altro indirizzo, naturalmente quello è l'indirizzo che dovremmo usare con SYS. La fig. 9.8 mostra l'assemblaggio MIKRO scritto in modo che possiamo leggere i codici. Il codice può essere assemblato a ogni indirizzo così, se vogliamo, possiamo trasformare questo in programma POKE di BASIC.

Ora c'è molto poco di nuovo da imparare, escluso ciò che riguarda le sorprese che il C64 riserva. Più si va avanti, più si accumuleranno esperienze fino a scoprire che le sorpese sono poche e facili. Quando arriverà quel momento, vi qualificherete come esperti, veri programmatori di codice macchina.

Appendice A

Come sono immagazzinati i numeri

Il C64 usa cinque byte di memoria per immagazzinare ogni numero. Se non abbiamo predisposizione per la matematica non possiamo essere molto bravi.

Per cominciare, i numeri (non interi) sono immagazzinati nella forma esponente mantissa. Questa è una forma usata anche per i numeri decimali. Per esempio, possiamo scrivere il numero 216000 come $2{,}16 \times 10^5$ (alla quinta), o il numero 0,00012 come $1{,}2 \times 10^{-4}$ (alla meno quattro). Quando è usato questo modo di scrivere i numeri, la potenza (di dieci in questo caso) è chiamata *l'esponente* e il moltiplicatore (un numero maggiore di 0 e minore di 1) è chiamato *mantissa*. Anche i numeri binari possono essere scritti in questo modo, ma con alcune differenze. Per iniziare, la mantissa di un numero scritto in questa forma è sempre frazionaria, ma non è scritto nessun punto. Secondariamente, l'esponente è una potenza di due piuttosto che di dieci. Possiamo quindi scrivere il numero binario 10110000 come 1011E1000. Questo significa una mantissa di 1011 (immaginiamolo come 0,1011) e un esponente di 1000 (2 alla potenza 8 in decimale). Non c'è nessun vantaggio nello scrivere piccoli numeri in questo modo, ma per grandi numeri è un vantaggio considerevole. Il numero

110101000000000000000000

per esempio può essere scritto come 110101E11000 (pensiamolo come 110101×2^{24}).

Questo schema è adatto al C64 e ad altre macchine che usano il Microsoft BASIC. Dato che il numero semplice più significativo della mantissa (la parte frazionaria del numero) è sempre 1, quando un numero è messo in questa forma è posto a 0 per dei propositi di immagazzinamento. L'esponente decimale, poi, ha 128 aggiunte prima di essere im-

magazzinato. Questo permette ai numeri con esponenti negativi fino a —128 di essere memorizzati senza complicazioni dato che un esponente negativo è immagazzinato come un numero il cui valore è minore di 128 decimale.

Il C64 usa quattro byte per immagazzinare la mantissa di un numero e un byte per l'esponente. Per fare un semplice esempio, consideriamo come il numero 20 (decimale) sarebbe codificato. Questo si converte in binario come 10100 che è $0{,}10100000 \times 2^5$ scrivendolo con il punto binario mostrato, usando otto byte e con l'esponente in forma decimale.

L'msb della frazione è poi cambiato in 0 in modo che il numero fornito sia 00100000. Quindi, con questa memoria, produrremo il numero (decimale) 32 nel byte più basso di mantissa. Nel frattempo l'esponente di 5 è 101 in binario. Il decimale 128 è aggiunto a questo per fare 10000101. Questa memoria vi darà 133 (che è 128+5).

Gli interi necessitano solo di due byte per la memorizzazione. L'intero deve avere un valore tra —32768 e +32767. Per cambiare un intero nella forma in cui il C64 lo immagazzina, procediamo come segue:

1. Se il numero è negativo: sottrarlo da 65536 e usare il risultato.
2. Dividere il numero per 256 e prendere la parte intera. Questo è il byte più importante.
2. Sottrarre dal numero 256× (byte più importante). Questo dà il byte meno importante.

Esempio: Cambiare 9438 nella forma intera di memorizzazione. Il numero è positivo, perciò può essere usato direttamente.

9438/256=36.867187.

L'mbs è perciò 36.

36*256=9216 e 9438—9216=222.

Il byte basso è 222.

Appendice B

Trasformazioni di esa e decimali

a) Esa in decimale

Per singoli byte (due numeri esa semplici)

Moltiplicare il numero semplice più importante per 16 e aggiungere l'altro numero semplice. *Per esempio:*

$3D è 3*16+13=61 decimale.

Per byte doppi (numeri di indirizzo)

Scrivere il numero semplice meno importante. Ora scrivere sotto di esso il valore del successivo numero semplice moltiplicato per 16. Sotto quello scrivere il successivo numero semplice moltiplicato per 256. Sotto ancora, scrivere il numero semplice successivo moltiplicato per 4096.

Per esempio:

$F3DB converte come segue:

Scrivere 1 numero semplice	11
Il successivo numero *16 è 13*16	208
Il successivo numero *256 è 3*256	768
Il successivo numero *4096 è 15*4096	61440
Il totale è	62427

Decimale in esa

Per singoli byte (minori del decimale 256)

Dividere per 16. La parte intera del numero è il numero semplice più importante. Il numero semplice meno importante è la parte frazionale del risultato moltiplicato per 16. Per esempio:

Per convertire 155 in esa:
153/16=9.6875, così 9 è il numero più importante.
Il numero meno importante è 6875*16, cioè 11. Questo si converte in esa B, così il numero è $9B.

Per numeri doppi - byte (numeri tra 256 e 65535 decimali)
Dividere per 16 come prima. Notate la parte intera del risultato e scrivete la parte frazionale 16 volte come un numero semplice esa. Ripetete il procedimento con la parte intera del numero fino a che non rimane un singolo numero semplice esa.
Per esempio:

Per trasformare 23815 in esa:
23815/16=1488,4375. La frazione 4375*16 dà 7 e questo è il numero semplice meno significativo.
Prendere il numero intero 1488/16=93.00. Dato che non c'è una frazione il successivo numero semplice è 0.
93/16=5,8125. La frazione 0,8125 moltiplicata per 16 dà 13, che è l'esa D. Questa è la terza figura esa. Dato che il numero intero è minore di 16 (è 5) allora è il numero semplice più importante e il numero intero è $5D07.

Appendice C

La serie delle istruzioni

La serie delle istruzioni del 6502 è relativamente breve, ma ci sono alcune istruzioni che non servono finché non si arriva a programmazioni molto avanzate. La descrizione completa di ogni istruzione prenderebbe troppo tempo e perciò l'abbiamo indicata con abbreviazioni. Per una descrizione dettagliata ricorrete ad uno dei libri che si riferiscono alla programmazione 6502. In generale M significa un byte in un indirizzo di memoria e i registri sono riferiti in modo non particolareggiato. Una freccia indica dove è immagazzinato il risultato di un procedimento. Per esempio $A+M+C \rightarrow A+C$ significa che il byte in memoria (indirizzato dall'istruzione) è aggiunto al byte nell'accumulatore A più il riporto C, e il risultato è posto nell'accumulatore A con un altro possibile riporto in C.

Il codice delle istruzioni è messo in colonne graduate secondo il metodo di indirizzamento. Questi metodi sono: Immediato, Pagina Zero, Pagina Zero × indicizzato, Assoluto, Assoluto × indicizzato, Assoluto Y indicizzato, Indiretto ×, Indiretto Y e PC relativo. Poche istruzioni usano poi Implied Addressing (Indirizzamento implicito). Questo metodo significa che non occorre nessun indirizzamento speciale. Nelle forme di linguaggio assembly, ADDR significa un indirizzo di due byte e addr significa un singolo indirizzo byte (più basso) per indirizzamento pagina 0. Disp. è usato per significare un dislocamento nell'Indirizzamento relativo PC. Byte significa il byte che segue un codice immediato di indirizzi. S è stato usato per significare il registro Processor Status. I flag (segnalatori) si riferiscono a C.N. e V. Tutti i codici OP sono mostrati in esa.

Forma del linguaggio Assembly	*Metodo di indirizzamento*	*Codice operativo*	*Azione*
ADC # Byte	Immediato	69	A+M+C→A+C
ADC addr	Pagina zero	65	
ADC addr, X	Pagina zero X	75	
ADC ADDR	Assoluto	6D	
ADC ADDR,X	Assoluto, X	7D	
ADC ADDR, Y	Assoluto, Y	79	
ADC (addr, X)	Indiretto, X	61	
ADC (addr), Y	Indiretto, Y	71	
AND # Byte	Immediato	29	A AND M→A
AND addr	Pagina zero	25	
AND addr, X	Pagina zero, X	35	
AND ADDR	Assoluto	2D	
AND ADDR,X	Assoluto, X	3D	
AND ADDR, Y	Assoluto, Y	39	
AND (addr, X)	Indiretto, X	21	
AND (addr), Y	Indiretto, Y	31	
ASL A	Implicito	0A	Shift sinistro
ASL addr	Pagina zero	06	
ASL addr, X	Pagina zero, X	16	
ASL ADDR	Assoluto	0E	
ASL ADDR, X	Assoluto, X	1E	
BCC Disp	Relativo	90	Salta se C=0
BCS Disp	Relativo	B0	Salta se C=1
BEQ Disp	Relativo	F0	Salta se Z=1
BIT addr	Pagina zero	24	OR con M
BIT ADDR	Assoluto	2C	Test N & V
BMI Disp	Relativo	30	Salta se N=1
BNE Disp	Relativo	D0	Salta se Z=0
BPL Disp	Relativo	10	Salta se N=0
BRK	Implicito	00	Interrompe il programma
BVC Disp	Relativo	50	Salta se V=0
BVS Disp	Relativo	70	Salta se V=1
CLC	Implicito	18	Cancella il riporto

Forma del linguaggio Assembly	*Metodo di indirizzamento*	*Codice operativo*	*Azione*
CLD	Implicito	D8	Cancella il modo decimale
CLI	Implicito	58	Cancella la disattivazione di interrupt
CLV	Implicito	B8	Cancella il flag V (segnalatore)
CMP #Byte	Immediato	C9	A—M, imposta i flag
CMP addr	Pagina zero	C5	
CMP addr, X	Pagina zero, X	D5	
CMP ADDR	Assoluto	CD	
CMP ADDR, X	Assoluto, X	DD	
CMP ADDR, Y	Assoluto, Y	D9	
CMP (addr, X)	Indiretto, X	C1	
CMP (addr), Y	Indiretto, Y	D1	
CPX # Byte	Immediato	E0	X—M impostano i flag
CPX addr	Pagina zero	E4	
CPX ADDR	Assoluto	EC	
CPY # Byte	Immediato	C0	Y—M impostano i flag
CPY addr	Pagina zero	C4	
CPY ADDR	Assoluto	CC	
DEC addr	Pagina zero	C6	M—1→M
DEC addr, X	Pagina zero, X	D6	
DEC ADDR	Assoluto	CE	
DEC ADDR, X	Assoluto, X	DE	
DEX	Implicito	CA	X—1→X
DEY	Implicito	88	Y—1→Y
EOR # Byte	Immediato	49	A EOR M>A
EOR addr	Pagina zero	45	
EOR addr, X	Pagina zero, X	55	
EOR ADDR	Assoluto	4D	
EOR ADDR, X	Assoluto, X	5D	
EOR ADDR, Y	Assoluto, Y	59	
EOR (addr, X)	Indiretto, X	41	
EOR (addr), Y	Indiretto, Y	51	
INC addr	Pagina zero	E6	M+1→M
INC addr, X	Pagina zero, X	F6	
INC ADDR	Assoluto	EE	
INC ADDR, X	Assoluto, X	FE	

Forma del linguaggio Assembly	*Metodo di indirizzamento*	*Codice operativo*	*Azione*
INX	Implicito	E8	X+1→X
INY	Implicito	C8	Y+1→Y
JMP ADDR	Assoluto	4C	Salta a ADDR
JMP (ADDR)	Indiretto	6C	Salta all'indirizzo immagazzinato
JSR ADDR	Assoluto	20	Salta alla subroutine
LDA # Byte	Immediato	A9	M→A
LDA addr	Pagina zero	A5	
LDA addr, X	Pagina zero, X	B5	
LDA ADDR	Assoluto	AD	
LDA ADDR, X	Assoluto, X	BD	
LDA ADDR, Y	Assoluto, Y	B9	
LDA (ADDR, X)	Indiretto, X	A1	
LDA (ADDR), Y	Indiretto, Y	B1	
LDX # Byte	Immediato	A2	M→X
LDX addr	Pagina zero	A6	
LDX addr, Y	Pagina zero, Y	B6	
LDX ADDR	Assoluto	AE	
LDX ADDR, Y	Assoluto, Y	BE	
LDY # Byte	Immediato	A0	M→Y
LDY addr	Pagina zero	A4	
LDY addr, X	Pagina zero, X	B4	
LDY ADDR	Assoluto	AC	
LDY ADDR, X	Assoluto, X	BC	
LSR A	Implicito	4A	Shift destro
LSR addr	Pagina zero	46	
LSR addr, X	Pagina zero, X	56	
LSR ADDR	Assoluto	4E	
LSR ADDR, X	Assoluto, X	5E	
NOP	Implicito	EA	Nessuna operazione
ORA # Byte	Immediato	09	A OR M→A
ORA addr	Pagina zero	05	
ORA addr, X	Pagina zero, X	15	
ORA ADDR	Assoluto	0D	
ORA ADDR, X	Assoluto, X	1D	
ORA ADDR, Y	Assoluto, Y	19	
ORA (addr, X)	Indiretto, X	01	
ORA (addr), Y	Indiretto, Y	11	
PHA	Implicito	48	Spinge A sullo stack (pila)

Forma del linguaggio Assembly	*Metodo di indirizzamento*	*Codice operativo*	*Azione*
PHP	Implicito	08	Spinge S sullo stack
PLA	Implicito	68	Preleva A dallo stack
PLP	Implicito	28	Preleva S dallo stack
ROL A	Implicito	2A	Ruota a sinistra
ROL addr	Pagina zero	26	
ROL addr, X	Pagina zero, X	36	
ROL ADDR	Assoluto	2E	
ROL ADDR, X	Assoluto, X	3E	
ROR A	Implicito	6A	Ruota a destra
ROR addr	Pagina zero	66	
ROR addr, X	Pagina zero, X	76	
ROR ADDR	Assoluto	6E	
ROR ADDR, X	Assoluto, X	7E	
RTI	Implicito	40	Ritorna dall'interruzione
RTS	Implicito	60	Ritorna dalla subroutine
SBC # Byte	Immediato	E9	A—M—C*→M
SBC addr	Pagina zero	E5	C* è un prestito
SBC addr, X	Pagina zero, X	F5	
SBC ADDR	Assoluto	ED	
SBC ADDR, X	Assoluto, X	FD	
SBC ADDR, Y	Assoluto, Y	F9	
SBC (addr, X)	Indiretto, X	E1	
SBC (addr), Y	Indiretto, Y	F1	
SEC	Implicito	38	Imposta il flag di riporto
SED	Implicito	F8	Imposta il modo decimale
SEI	Implicito	78	Imposta la disattivazione di interrupt
STA addr	Pagina zero	85	A→M
STA addr, X	Pagina zero, X	95	
STA addr	Assoluto	8D	
STA ADDR, X	Assoluto, X	9D	

Forma del linguaggio Assembly	*Metodo di indirizzamento*	*Codice operativo*	*Azione*
STA ADDR, Y	Assoluto, Y	99	
STA (addr, X)	Indiretto, X	81	
STA (addr), Y	Indiretto, Y	91	
STX addr	Pagina zero	86	X→M
STX addr, Y	Pagina zero, Y	96	
STX ADDR	Assoluto	8E	
STY addr	Pagina zero	84	Y→M
STY addr, X	Pagina zero, X	94	
STY ADDR	Assoluto	8C	
TAX	Implicito	AA	A→X
TAY	Implicito	A8	A→Y
TYA	Implicito	98	Y→A
TSX	Implicito	BA	S→X
TXA	Implicito	8A	X→A
TXS	Implicito	9A	X→S

Appendice D

Metodi di indirizzamento del 6502

Ogni metodo di indirizzamento ha l'effetto di usare un byte nella memoria. L'indirizzo al quale questo byte è memorizzato è chiamato L'INDIRIZZO EFFETTIVO (EA). Il proposito di ogni metodo di indirizzamento è fare uso di un indirizzo effettivo.

Indirizzamento immediato: l'EA è l'indirizzo che segue immediatamente l'istruzione byte.

Indirizzamento pagina Zero: solamente il byte più basso dell'EA è dato nell'istruzione. Il byte superiore è sempre 00 donde il nome pagina 0. Per esempio, usando l'indirizzamento pagina 0 con un byte di FB dovremmo fare uso dell'indirizzo $00FB.

Indirizzamento Assoluto: L'istruzione è seguita da due byte che formano un indirizzo completo. Per esempio LDA $563F significa che l'accumulatore deve essere caricato all'indirizzo $563F.

Indirizzamento indicizzato: un numero è memorizzato in uno dei registri dell'indice (X o Y). L'indirizzo effettivo è il numero più ogni indirizzo specificato nell'istruzione. Per esempio LDA 25,X significa caricare l'accumulatore dall'indirizzo che consiste del numero memorizzato nel registro X più 25.

Indirizzamento implicito: l'indirizzo è implicito nell'istruzione e non occorre nessun indirizzo speciale. Per esempio INX significa incrementare il registro X e non occorre alcun EA.

Indirizzamento indiretto: Ci sono due forme, ed entrambe usano indirizzi di pagina zero. La indiretta X-indicizzata aggiunge il contenuto del registro X al numero di indirizzo di pagina zero e va a cercare il byte a questo indirizzo. Questa forma il byte basso dell'indirizzo effettivo, poi il byte alto si va a cercare dal successivo indirizzo maggiore di pagina zero. L'indiretta Y-indicizzata va a cercare il byte dell'indirizzo di pagina zero e poi aggiunge i contenuti del registro Y a questo byte. Questo è il byte più basso dell'indirizzo effettivo e il byte più alto si va a cercare dall'indirizzo successivo di pagina zero come prima.

Appendice E

Alcuni indirizzi della ROM e della RAM

Ora che è disponibile il disassemblaggio completo per la ROM del C64, tutti gli indirizzi di questa memoria saranno a vostra conoscenza. Assieme ad una scelta degli indirizzi più utili sono stati inclusi alcuni indirizzi della RAM, seguiti da brevi note esplicative sul loro contenuto.

Indirizzi della ROM

Routine INPUT $F157
Routine OUTPUT $F1CA
Lettura tastiera $E112
Stampa su schermo $E10C
Inizio BASIC $A000
READY messaggio $A376
NEW $A644
Esecuzione istruzione $A7E4
Chip di controllo del video da $D000 a $D021
Port 1 da $DC00 a $DC0F
Port 2 da $DD00 a DDOF

Indirizzi della RAM

$2B,$2C	Inizio BASIC
$2D,$2E	Inizio VLT
$2F,$30	Inizio Matrici
$31,$32	Fine Matrici
$33,$34	Inizio memorizzazione stringa

\$37,\$38	Quantità memoria
\$9A	Codice di output periferica
\$C5	Ultimo tasto premuto
\$CC	Abilità cursore (0=lampeggio)
\$D1,\$D2	Linea schermo attuale
\$D3	Posizione del cursore
\$D6	Numero di linea cursore
\$F3,\$F4	Memoria del colore
\$0200	Buffer input BASIC
\$0277	Buffer di tastiera
\$028A	Repeat di tastiera (\$80 estende la funzione a tutta la tastiera)
\$028F,\$0290	Indirizzo routine di decodifica tastiera.

DELLO STESSO EDITORE

CODICE ISBN	TITOLO	PREZZO

COLLANA INFORMATICA

1204	*PROGRAMMARE CON IL LINGUAGGIO ADA*	L. 16.000
1174	*ANALISI E PROGRAMMAZIONE STRUTTURATA. CONCETTI FONDAMENTALI*	L. 9.000
1557	*APPLE // - IL LOGO*	L. 19.000
1565	*APPLE // - UNA GUIDA FACILE*	L. 15.000
1646	*APPLE // - 35 PROGRAMMI IN BASIC LISTATI + DISCO*	L. 26.000
1514	*APPLE // 72 PROGRAMMI + DISCO*	L. 29.500
1247	*ASSEMBLER PROGRAMMARE IN ASSEMBLER*	L. 13.000
0879	*BANCA DATI. APPLICAZIONI SUI PICCOLI E GRANDI CALCOLATORI*	L. 18.000
1492	*BASIC. CORSO DI PROGRAMMAZIONE PER MICROCALCOLATORI - (CHIP SPECIAL)*	L. 8.000
1220	*BASIC FACILE*	L. 16.000
0887	*BASIC. PROGRAMMAZIONE DEI MICROCALCOLATORI*	L. 8.000
1239	*CAD IMPIEGO DEI SISTEMI DI PROGETTAZIONE ASSISTITA DA CALCOLATORE*	L. 25.000
159X	*COME GESTIRE MEGLIO IL C 64*	L. 22.000
1832	*COME VINCERE CON I VIDEOGIOCHI*	L. 12.000
1581	*COMMODORE VIC 20 GIOCAEVINCI CON CASSETTA*	L. 14.000
1670	*COMMODORE VIC 20 GUIDA - LIBRO + CASSETTA*	L. 19.000
1743	*COMMODORE VIC 20 IL LIBRO DEI GIOCHI*	L. 14.000
8039	*COMMODORE VIC 20 SOFT 1 (CASSETTA)*	L. 8.000
1794	*COMMODORE VIC 20 1° (CASSETTE)*	L. 44.000
1476	*COMMODORE VIC 20 1° LISTATI*	L. 6.000

CODICE ISBN	TITOLO	PREZZO
8012	*COMMODORE VIC 20 2° (CASSETTE)*	L. 25.000
2049	*COMMODORE VIC 20 2° LISTATI*	L. 7.000
2057	*COMMODORE 16 MANUALE D'USO*	L. 18.000
808X	*COMMODORE 64 AVVENTURIERO (DISCHETTO)*	L. 22.000
1662	*COMMODORE 64 GUIDA ALL'USO*	L. 15.000
8063	*COMMODORE 64 HOBBY ELETTRONICA (DISCHETTO)*	L. 22.000
8098	*COMMODORE 64 PARTY-NONSENSE (DISCHETTO)*	L. 22.000
1603	*COMMODORE 64 PROGRAMMAZIONE SEMPLICE*	L. 12.000
8071	*COMMODORE 64 ROMPICAPO (DISCHETTO)*	L. 22.000
8055	*COMMODORE 64 STUDIO E SCIENZA (DISCHETTO)*	L. 22.000
1719	*COMMODORE 64 1° PROGRAMMI REGISTRATI SU CASSETTA*	L. 25.000
8004	*COMMODORE 64 1° PROGRAMMI REGISTRATI SU DISCHETTO*	L. 25.000
1999	*COMMODORE 64 2° - PROGRAMMI LISTATI*	L. 7.000
8160	*COMMODORE 64 2° PROGRAMMI REGISTRATI SU CASSETTA*	L. 25.000
8152	*COMMODORE 64 2° PROGRAMMI REGISTRATI SU DISCO*	L. 25.000
8047	*COMMODORE 64/SPECTRUM SOFT 2 (CASSETTA)*	L. 8.000
8101	*COMMODORE 64/SPECTRUM SOFT 3 (CASSETTA)*	L. 8.000
811X	*COMMODORE 64/SPECTRUM SOFT 4 (CASSETTA)*	L. 8.000
8136	*COMMODORE 64/SPECTRUM SOFT 5 (CASSETTA)*	L. 8.000
114X	*DISEGNARE COL COMPUTER*	L. 10.000

CODICE ISBN	TITOLO	PREZZO
1409	FORTH. CORSO DI PROGRAMMAZIONE PER MICROCALCOLATORI	L. 12.000
1689	FORTRAN IV	L. 21.000
1859	GRAFICA COL COMPUTER	L. 6.000
1867	GRAFICA TRIDIMENSIONALE PER IL PERSONAL COMPUTER	L. 12.000
1905	IBM PC APPLICAZIONI PROFESSIONALI	L. 32.000
1778	IBM PC FOGLI ELETTRONICI	L. 28.000
176X	IBM PC GESTIONE DEGLI ARCHIVI DI DATI	L. 29.000
8128	IBM PC GESTIONE DEGLI ARCHIVI DI DATI (DISCO)	L. 30.000
1808	IBM PC GESTIONE DEI FILE	L. 17.000
1468	IBM PC GRAFICA	L. 24.000
8144	IBM PC GRAFICA (DISCO)	L. 30.000
1336	IBM PC GUIDA ALL'IMPIEGO	L. 25.000
145X	IBM PC IMPIEGO DEL CP/M-86	L. 22.000
1573	IBM PC INTRODUZIONE	L. 16.000
1735	IBM PC jr INTRODUZIONE	L. 16.000
1506	IBM PC SISTEMA OPERATIVO DOS	L. 28.000
1638	IBM PC VISICALC	L. 27.000
1654	IBM PC 35 PROGRAMMI IN BASIC LISTATI + DISCO	L. 26.000
1840	INFORMATICA IN FABBRICA	L. 24.000
0445	INTRODUZIONE AI MICROPROCESSORI E AI MICROELABORATORI	L. 14.000
1611	INTRODUZIONE AI PERSONAL COMPUTER	L. 10.000
1484	INTRODUZIONE ALL'INFORMATICA	L. 14.000
1980	MACINTOSH. COME E DOVE USARLO	L. 20.000
2073	MACINTOSH. GUIDA ALL'USO	L. 32.000
1883	MANUALE D'USO LOTUS 1-2-3	L. 26.000
1328	PASCAL. CORSO DI PROGRAMMAZIONE	

CODICE ISBN	TITOLO	PREZZO
	PER MICROCALCOLATORI	L. 16.000
1522	*PASCAL VELOCE*	L. 16.000
064X	*PROGETTARE CON I BONDGRAPH*	L. 14.000
0844	*TELEMATICA*	L. 42.000
1441	*UNIX INTRODUZIONE AL SISTEMA OPERATIVO PER MICROCALCOLATORI*	L. 20.000
162X	*WORDSTAR*	L. 18.000
1751	*ZX SPECTRUM GIOCARE CON (LIBRO + CASSETTE)*	L. 12.000
1891	*ZX SPECTRUM GUIDA ALL'IMPIEGO*	L. 18.000
1816	*ZX SPECTRUM PROGRAMMI LISTATI*	L. 6.000
1344	*ZX 81-SPECTRUM 84 PROGRAMMI LISTATI*	L. 6.000

Prima o poi la maggior parte degli utenti Commodore 64 si sentono limitati nel linguaggio BASIC. Diventa quindi necessario imparare ad usare il codice macchina per un'operatività più immediata e per il completo controllo del computer: creare effetti sonori, animazione multipla, divisione dello schermo. Con il codice macchina, le vostre istruzioni esercitano un controllo diretto sul microcomputer, bypassando i comandi BASIC.

In questo libro troverete quello che occorre per iniziare; la sua comprensione presuppone solo una discreta conoscenza del BASIC. Imparerete che cosa è il codice macchina, come funziona, come immettere i codici, far girare i programmi e salvarli. Acquisterete, di conseguenza, una conoscenza più ''intima'' e più gratificante del vostro micro. La presentazione della materia è trattata in modo tale da trasmettere al lettore la spinta occorrente per dei progressi autonomi in questo affascinante campo.

Il codice macchina amplierà il vostro ''repertorio'' esaltando le possibilità del vostro Commodore 64.

L. 19.000 (IVA inclusa) **ISBN 88 7081 1786**

COMMODORE 64 - CODICE MACCHINA

I. Sinclair